

## Gamry Framework™: Overview

The Gamry Framework™ is the basis for all Gamry Instruments Windows based software. All application program functions are accessed through the Framework menus.

The Framework is the program that runs when you select FRAMEWORK from the Windows Start Menu or click the Framework icon on your desktop. It makes a menu structure, editing tools, and the Explain™ Experimental Control language available to applications such as the DC105™ DC Corrosion Measurement System and the EIS300™ EIS System. One Framework can be shared by several Gamry Instruments applications.

You will use the Gamry Framework for three basic purposes: running experiments, editing experimental scripts and starting up the Gamry Echem Analyst™ software.



The [Experiment](#) pull down menu gives you the ability to run a standard script from any loaded software system (EIS300™, DC105™, PHE200™, etc.), to run a previously developed "named script", or to run the script currently being edited. Each experiment resides in a "Runner" window as it runs.

The [File](#) and [Edit](#) pull down menus give you the commands necessary for the generation and editing of experimental scripts. The Framework editor was intended for use with scripts written in the Explain Experimental Control Language, but you can also use it for other small editing jobs. A selection on the [File](#) menu also allows you to calibrate your potentiostat(s).

The [Analysis](#) pull down menu allows you to start the Gamry Echem Analyst, either by itself, or with a recently run data file.

The [Options](#) menu gives you the ability to customize your version of the Gamry Framework.

The [Window](#) menu gives you the tools to manage multiple processes. For example, you can arrange your computer's display to show the data being acquired in a Runner window while you simultaneously edit an experimental script.

Finally, the [Help](#) menu gives you access to this Help information. It also gives access to revision information about your copy of the Gamry Framework.

## Runner Windows

All experiments are started up via the Framework's Experiment pull down menu. Once you select a standard or custom script from this menu, the Framework creates a [Runner window, and opens up a Setup dialog box within that window](#). The [contents of a Runner window change](#) as the experiment proceeds, but the window always remains associated with a single experiment and the script running in that window.

You can have more than one Runner window open at a time. Each Runner window runs one experiment (by running its associated Explain script). The Runner window stays open as long as the script is running. The Runner windows can be resized, moved around, and reduced to an icon. You control the display of Runner windows via the [Window](#) menu.

Each Runner window can have one or more potentiostats associated with it. When a potentiostat is associated with one window, it cannot be selected for use by another until its owner has closed the potentiostat. In general, the first window must be closed before the potentiostat is available for use in another window.

Similarly, a Runner window generally has an output file reserved for its data storage. An output file opened by one window must be closed before it is available for use by any other window or program. Most Framework scripts keep their data file open throughout data acquisition.

A Runner window can open and close several output files as its script executes. It can have only one file open at a time.

You can get the value of a data point by pointing the mouse at the point and clicking the left mouse button. The X and Y coordinates of the mouse pointer tip appear at the upper left corner of a real time data display.

## Runner Window Controls

Three push button controls, labeled F1 through F3, are used to control the process taking place in the Runner window. An additional control called the Curve List, is associated with the F5 key. The Curve List includes a text field, often showing a label ACTIVE, and a small push button labeled with a downward pointing arrow.

The [Runner window](#) always displays all four controls. The control buttons are each assigned to a function key as well as to the button on the display. An example is the **F1-ABORT** button, which can be selected by clicking on it with the mouse, or by hitting the F1 function key.

Some of the Runner window control assignments can change during the experimental sequence. For example, F3 is assigned to **Pause** while an experiment is running and to **Continue** while an experiment is paused. Any button that has no function assigned to it will be unlabeled and grayed out.

 [Skip](#)

 [Abort](#)

 [Pause](#)

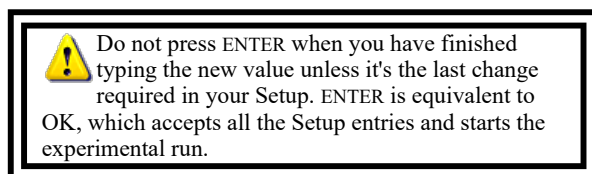
 [Continue](#)

 [Curve List](#)

## Entering and Changing Setup Window Parameters

The Setup dialog box is used to change the parameters and switches that the technique will use in running an experiment. Accessing a parameter in the Setup dialog box can be done using either the keyboard or the mouse.

From the keyboard, you can use the TAB key to advance down the Setup dialog box. The parameter that is currently selected is highlighted in reverse video or with a dotted box. Once the parameter that you wish to edit is active, you can replace it by typing a new value. The old value will be erased as soon as you press a printing character. If you wish to retain and edit the old parameter, press a cursor movement key (such as a DIRECTION key or HOME) before the first printable key.



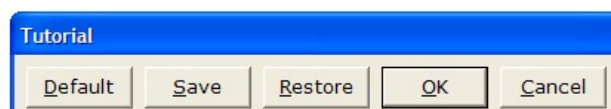
If you're using the mouse to access a parameter, just click the mouse somewhere within the value. You will get an editing cursor in the value. Use the DELETE, BACKSPACE and DIRECTION keys to move around in and erase the old parameter value. Any new digits that you type will be inserted after the current location of the editing cursor. Do not press ENTER when you have finished typing the new value unless it's the last change required in your Setup. ENTER is equivalent to OK, which accepts all the Setup entries and starts the experimental run.

Most numerical values are entered as floating point values. We recommend that you only enter a decimal point in an entered number when one is required. For example, enter 1 not 1.0. If you need to enter very large or very small values, E format can be used. 1.23E-4 equals 0.0001234. Note that E (upper and lower case) are the only valid letters that can be entered in a number. Unless otherwise noted, all values in the standard techniques are entered in SI values.





On/off switches in Setup are indicated by square check boxes. With the mouse you alter the state of a switch by clicking on the box. From the keyboard TAB down until the switch state is enclosed in a dotted line, then press SPACE to change the state. The label for a switch always indicates the current state of the switch, not the state you will get after switching.

## Setup Control Buttons

When you setup one of the standard experiments, you will have 5 buttons available at the top of the Setup dialog box. These 5 controls are independent of which technique you are setting up. They are used to control the Setup box itself, or to control the saving/recalling of parameter sets from your computer's disk.



 [Default](#)

-  [Save](#)
-  [Restore](#)
-  [OK](#)
-  [Cancel](#)

## Editor Windows

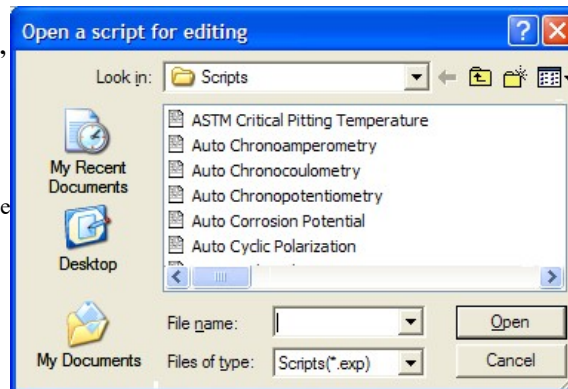
An Editor window is created when you select **File, New Editor** or **File, Open...** from the Framework menu. **File, New Editor** opens a blank editor window. **File, Open...** pops up a dialog box asking for the name of the file to edit.

An Editor window displays text that can be modified. You can edit several files simultaneously. Each file will reside in its own Editor window.

*NOTE:* You can also open up two or more Editor windows to edit one file! While this is a bit dangerous, it is a feature, not a software bug. Different windows allow you to see different portions of the file simultaneously. They also can make cut and paste operations simpler. If you do make use of this feature, remember to use one Editor window as the master. Only save the copy in this master window to disk.

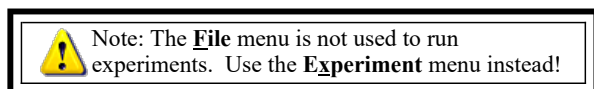
The Editor windows can be moved around, resized, or minimized. You can control the display of all Framework windows via the **Window** menu.

Whenever an Editor window is the active window, the **Edit** pull down menu on the Framework menu bar is activated. The standard cut, copy, and paste functions are available on this menu. The **Goto...**, **Find...**, **Next** and **Previous** commands allow you to move around in the text being edited.








## File Menu Commands

The selections on the **File** pull down menu deal mainly with the Framework's built-in script editor.



If you use only the standard experimental scripts supplied by Gamry Instruments, you never need to use the **File** commands associated with the Editor. The only **File** menu command that you will routinely use is **File, Exit**.

Most of the commands on the **File** menu are familiar windows commands. Similar commands are found in nearly every Windows program.

-  [New Editor](#)
-  [Open...](#)
-  [Save](#)
-  [Save All](#)
-  [Save As...](#)
-  [Close](#)
-  [Print...](#)
-  [Exit](#)

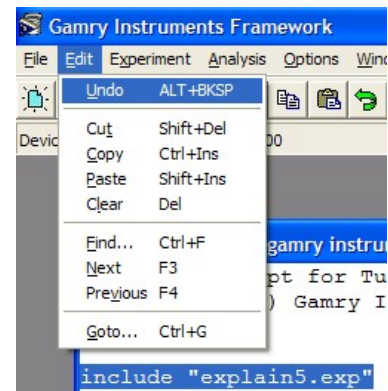


## Edit Menu Commands

The **E**dit menu commands provide simple Editor functions. The selector for the **E**dit menu is grayed out and inactive unless the active window is an Editor window.

The Edit pull down menu contains commands which work on selected text. To select a block of text with a mouse, hold down the left mouse button as you move the mouse cursor over the block. To select a block of text from the keyboard, hold down the SHIFT key as you move the cursor over the block with the arrow keys. In either case, selected text within the Framework editor is shown in reverse video.

-  [Undo](#)
-  [Cut](#)
-  [Copy](#)
-  [Paste](#)
-  [Clear](#)
-  [Find...](#)
-  [Next](#)
-  [Previous](#)
-  [Goto...](#)






## Experiment Menu Commands

A typical **E**xperiment pull down menu is shown to the right. The items found on this menu are not the same in all systems. They depend on:

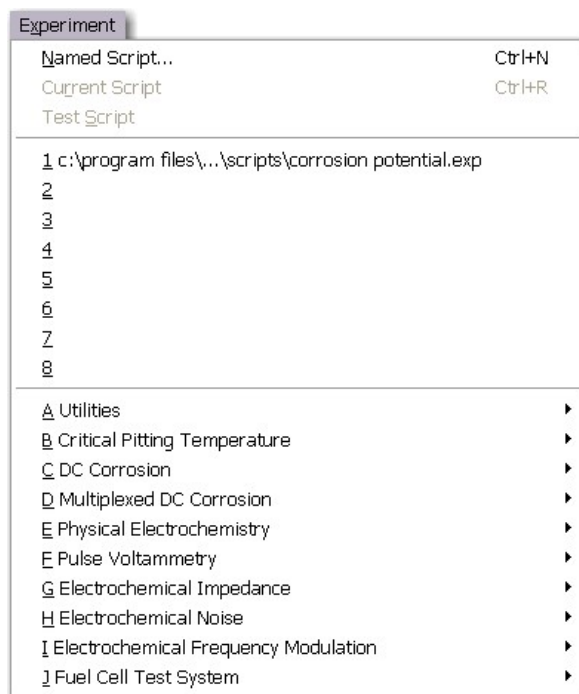
- Which Gamry Instruments Windows based software systems you have installed.
- The order in which you installed your software systems.
- The most recently run experiments.
- Custom modifications that have been done to the menu.

All of the commands in the **E**xperiment menu directly or indirectly open a Runner window and then compile and execute an experimental script in that window.

The **E**xperiment pull down menu is always divided into 3 sections. You use

-  [The bottom section of the menu to access standard techniques.](#)
-  [The middle section of the menu to rerun a recently used script.](#)
-  [The upper section of the menu to run a Named script.](#)

The **U**tilities sub-menu is always installed with Framework software. It contains links to the [calibration script](#) as well as other [various utilities](#) that users may find helpful.



## Standard Techniques

Each Gamry Instruments Windows software product adds an entry in the bottom section of the **E**xperiment pull down menu. When you select one of these entries, a [child menu](#) appears to the right of the pull down menu.

The child menu contains the names of standard techniques. Selecting a technique from the child menu opens a Runner window and then executes the script for that technique in that Runner window.



**NOTE:** A "file not found error" will occur if you select a standard technique and the corresponding "\*.EXP" file cannot be loaded. This is usually the result of the program looking in the wrong directory.

An advanced user can rename and/or add new child menus. He can also alter the contents of the child menus. This is done via entries in the "GAMRY.INI" file

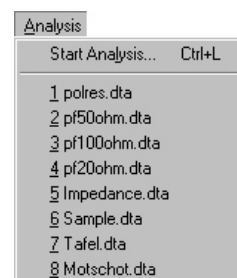
## Analysis Menu Commands

A typical **Analysis** pull down menu is shown to the right.

The **Analysis** menu is used to access the Gamry Echem Analyst. The menu is divided into two sections. The top section allows the user to start the Gamry Echem Analyst. The bottom section is a list of the most recently used (MRU) data files.

The **Start Analysis** command is used to start the Gamry Echem Analyst if it is not already open. No data file will be opened.

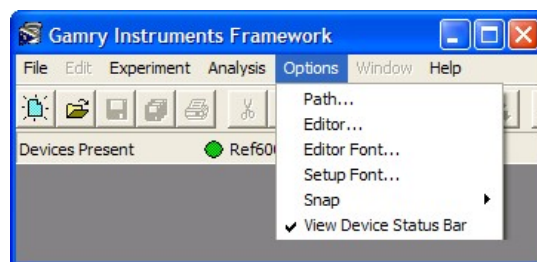
Clicking on the **MRU** list will start the Gamry Echem Analyst and open the data file selected from the **MRU**.



## Options Menu Commands

The commands under this menu are used to customize your Framework settings. Each preference is written to the GAMRY.INI file so each time Framework is started, your previous settings are recalled.

### Options Pull-Down Menu



 [Path](#)

 [Editor](#)

 [Editor Font](#)

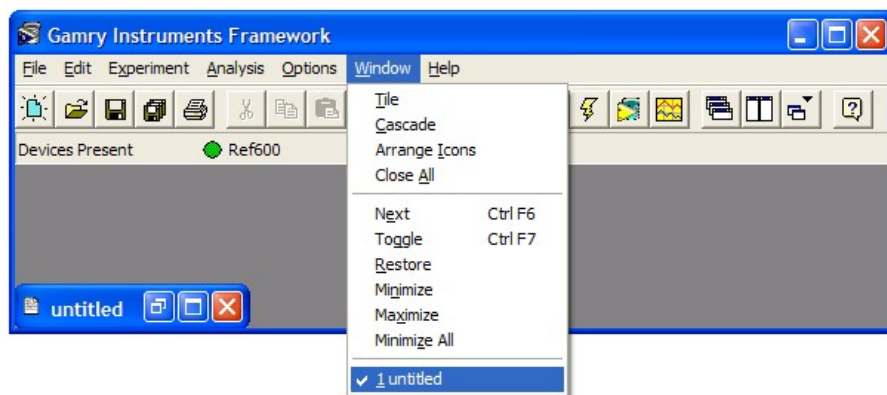
 [Setup Font](#)

 [Snap](#)

 [View Device Status Bar](#)

## Window Menu Commands

The **Window** menu is used to control the display of Editor and Runner windows within the Framework window. Technically, Editor and Runner windows are known in Windows terminology as "child" windows. The Framework itself runs in a "parent" window.



Framework child windows can be controlled via their window control menu. This menu is accessed via a button located at the upper left corner of the window. You can minimize, maximize, and close an open window using the commands available on the window control menu.

There are [2 icons](#) which symbolize minimized Framework windows. The icon for a minimized Editor window looks like a sheet of paper. The icon for a minimized Runner window looks like a chemist's beaker and an operational amplifier (shown as a triangle).

The Framework window can be resized and moved around. The Framework child windows are always displayed within the parent window. Dialog boxes are an exception. They are a fixed size and can exceed the boundaries of the Framework window.

The Window Menu selections are listed below:

- [Tile](#)
- [Cascade](#)
- [Arrange Icons](#)
- [Close All](#)
- [Next](#)
- [Toggle](#)
- [Restore](#)
- [Minimize](#)
- [Maximize](#)
- [Minimize All](#)
- [Window Names](#)

## Help Menu Commands

The Help menu allows the user to select several Online Help functions, as well as view the current version of their Framework software.

 [Contents](#)

 [Search](#)

 [About the Framework](#)

## Data File Overview

A Gamry Framework data file is a standard "tab delimited" ASCII file. By default it has a ".DTA" filename extension. You can use other filename extensions, but they will be less convenient.

The Windows Notepad can be used to examine and modify Gamry Framework data files, but with some care. Microsoft Excel can also read these data files, and can be used to view or print them. However, editing and saving a file using Microsoft Excel may render the file unreadable by the Gamry Echem Analyst.

The data file is a collection of data objects. Each data object starts on a new line and is identified by a **type**, a **tag**, and one or more **values**. The **type** is a text descriptor of the kind of object represented by this information. Any text at the beginning of a line is assumed to be a **type** identifier. The data objects can be of various **types**. The **type**, **tag**, and **values** of a data object are separated by the tab character.

After each **type** identifier, there is a **tag** identifier. This is a unique name for a specific object. Setup parameter data objects may also have one or more **values** following the **type** and **tag**.

The first two lines of a Framework data file are generally two special and important data objects.

**EXPLAIN  
TAG TYPICALEXPERIMENT**

The EXPLAIN *type* identifies this file as one output from an Explain Script. Without this line, a file will not be read in by the Gamry Echem Analyst.

The next line begins an object, uniquely identified as *type* TAG, that is used to specify which experiment created the data file. In this example, the *tag* is "TYPICALEXPERIMENT". This object is used by the Gamry Echem Analyst to determine which Analysis script should be used when this data file is opened.

Data objects are generally more complicated. In the following example, the object is of *type* "POTEN", and stores information about a potential. The *tag* or name of this object is "VINIT". The information about the poten is contained in the 3 fields that follow the *tag*. The tab character is represented by "<tab>" so you do not confuse it with the space character.

**POTEN<tab>VINIT<tab>-5.00000E-001<tab>F<tab>Initial &E (V)**

One special *type* is a TABLE. A TABLE is output by a data curve. The *tag* for a data curve is followed by the number of data points. The table headings, units, and data values follow on subsequent lines. The table headings and units help to identify the data that is stored in a column. In the example below, the fourth column stores the measured current (Im) and the unit used to store the data is the ampere (A=ampere). The current stored for point number 3 is about  $-1.13 \times 10^{-4}$  A, or 113  $\mu$ A.

**TABLE CURVE 21**

```
Pt T Vf Im Vu Sig Ach Over
# s V A V V V bits
0 0.99998 -4.98000E-001 -2.06420E-004 0.00000E+000 -5.00000E-001 1.54350E+000 .....
1 1.99995 -4.49000E-001 -1.56950E-004 0.00000E+000 -4.50000E-001 1.54350E+000 .....
2 2.99993 -3.98000E-001 -1.31810E-004 0.00000E+000 -4.00000E-001 1.54350E+000 .....
3 3.99999 -3.49000E-001 -1.13670E-004 0.00000E+000 -3.50000E-001 1.54360E+000 .....
```

Fields (*types*, *tags*, and *values*) within data objects are separated by the tab character. One exception is the separator between data points in a data curve, which is a carriage return.


Data objects can appear in any order in the file. The Gamry Echem Analyst locates objects within the file by means of their *tags*.

## Overloads

Some Gamry data files will have a column in the data table with the heading "Over" and units of "bits". This column will consist of a string of 13 characters. If no overloads were detected during the acquisition of a data point, all of these characters will be a period ("."). A character other than "." indicates an unusual and possibly serious overload condition existed during that data point. An example is shown below. As with all Tables, the columns are <tab> delimited.

```
OCVCURVE TABLE 40
Pt T Vf Vm Ach Over
# s V vs. Ref. V V bits
0 0.258 -4.70E-001 0.00E+000 1.84E-003 .....
1 0.516 -3.73E-001 0.00E+000 2.86E-003 techihssivarq
```

In this example, no overloads were detected for point #0, and each of the 13 characters is the "." character. In point #1, all of the possible overloads are set. The non-period character in each position indicates that an overload was detected and recorded. The position of the character in the string indicates the type of overload.



Note: The position of the character in the string indicates the type of overload, not the character. An "h" in the 4th position and an "h" in 6th position indicate different kinds of overload conditions!

Pos.	Char.	Meaning	Possible cause
1	t	Timing problem	Data rate is too fast
2	e	Potential Overload (Electrometer hardware limit exceeded)	Cell voltage is too big to measure
3	c	CA Overload (Control Amplifier hardware limit exceeded)	Potentiostat is oscillating or Potentiostat can

			not control the cell potential or current
4	h	CA Overload <i>sometime</i> during the data point ( <b>H</b> istory CA Overload)	Transient overload, CA speed not optimized
5	i	I Overload (I/E converter hardware limit exceeded)	Wrong I/E range
6	h	I Overload <i>sometime</i> during the data point ( <b>H</b> istory I Overload)	Current spike or transient
7	s	Settling problem, hardware. Some hardware relay or range had not settled to its final position when the data point was taken.	Experiment is too fast to autorange I/E at these current levels
8	s	Settling problem, software. Either the operator or the Explain script triggered a data point before the hardware was in a stable state.	Revise script or retrain operator
9	i	ADC input was out of range for a current measurement	Wrong I channel range or offset
10	v	ADC input was out of range for a voltage measurement	Wrong V channel range or offset
11	a	ADC input was out of range for a measurement of the auxiliary ADC input	Wrong Aux channel range or offset
12	r	Raw data overrun. Raw data queue is filled.	Computer is too busy or too slow for this experiment
13	q	Processed data overrun. Processing queue is filled.	Computer is too busy or too slow for this experiment

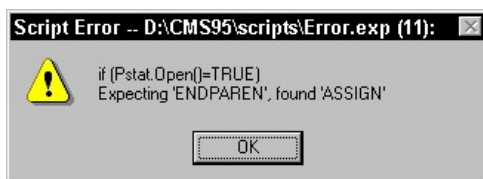
## Errors in Explain Scripts

There are two basic types of Explain errors that are detected by the Framework.

The first type is a compiler error. When a script is loaded into the Framework it is "tokenized". During this process the script is converted to a form that can be executed by the Framework's Explain interpreter. If the Framework finds a section of the script that it cannot tokenize, it reports a compiler error.

A compiler error is reported in a dialog box similar to the figure below.

Compile Error Dialog Box



We introduced an intentional syntax error into a script to produce this figure. The title of the error box tells you the name of the file in which the error occurred (ERROR.EXP). The number in parentheses after the filename is the line number on which the error was detected (line 11). In this example, the error was an incorrect operator. We wanted the equality operator "eq", but we entered the assignment operator "=".

The second type of error is a runtime error. It occurs as a script is being executed.

A runtime error is reported in a different dialog box, similar to the figure below.

RunTime Error Dialog Box





Again, we introduced an intentional runtime error into a script to produce this figure. The title of the error box tells you the name of the file in which the error occurred (ERROR.EXP). The number in parentheses after the filename is the line number on which the error was detected (line 11). In the example, the error was a typing mistake in line 11 of the code. "LABEL.New" was typed as "LABEL.Now". This is a runtime error rather than a compiler error, because the compiler that tokenizes the code assumed that "LABEL.Now" will be defined in a different code module.

The line number reported in either type of error box may not be exact. Continuation lines can confuse the line number counter. Also, the line number reported in an error box is the line number being compiled or run when the error was detected. The incorrect line that really caused the error may be earlier in the script.

You can use the **Edit, Goto...** command to quickly find the reported error line when you edit the file.

## Ru Estimation - Purpose

Electrochemical test cells always have a solution resistance controlled by the cell's geometry and the composition of the cell's electrolyte. Current flow through this solution resistance can cause significant errors in the cell's measured potential.

A three-electrode potentiostat compensates some of the cell's solution resistance through placement of the cell's reference electrode. Because reference electrodes are not infinitely small and they cannot be placed infinitely close to the working electrode, this compensation is always incomplete. The portion of the total solution resistance that is left uncompensated is called  $R_u$  (uncompensated resistance).

Positive feedback  $iR$  compensation can be used to correct for the effects of  $R_u$  - if the  $R_u$  value is known. In essence, positive feedback  $iR$  compensation circuitry, at a desired potential  $E_d$ , applies a potential  $E_d + iR_u$  (where  $i$  is the measured cell current). With this voltage applied between the working and reference electrodes, the electrochemical interface at the working electrode has a potential of  $E_d$ .

Unlike some other  $iR$  compensation methods, ideally current feedback  $iR$  compensation is fast, it is quiet, it dynamically corrects for the  $iR_u$  error as the cell current changes, and the actual voltage applied to the electrode interface is the same as the potentiostat's input voltage.

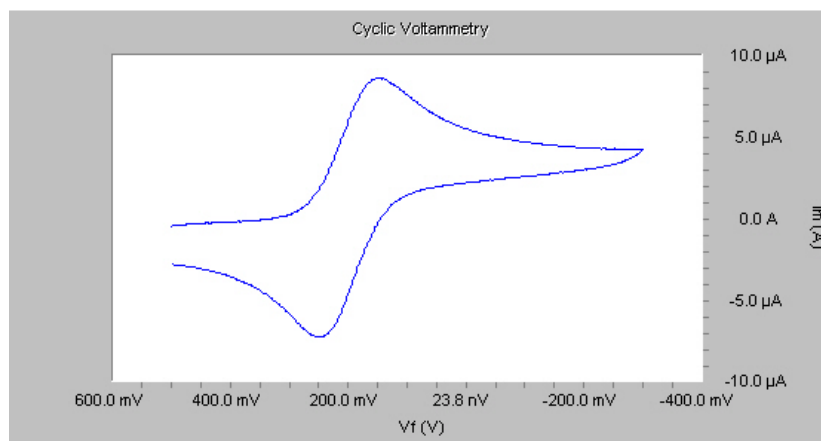
However, there are at least two problems with positive feedback  $iR$  compensation:

1. The value of  $R_u$  must be determined or calculated before the method can be applied.
2. Excessive compensation can cause instability and oscillation in the potentiostat's electronics.

The Utilities Menu includes a script that uses Electrochemical Impedance Spectroscopy (EIS) to make a determination of  $R_u$ . The script name is "GetRu.EXP". It is called from the Framework menu by selecting, Experiment, Utilities, Get Ru.

When the "GetRu.EXP" script runs, the user selects a potential, and the EIS determination is made at that potential. The user should carefully select this potential. In general, the potential used for  $R_u$  determination should be free from electrochemical reactions.

For example, the figure below shows the Cyclic Voltamogram of Ferrocyanide on a Platinum electrode in 0.1 M KCl. Notice the reduction peak beginning at +0.3 V vs. SCE. Above this voltage, no Faradiac current is seen. We would recommend that the  $R_u$  determination in this system is made at 0.5 V vs. SCE.



In EIS, a small sinusoidal voltage is applied to the cell. The resulting current should be a sinusoid at the same frequency, but different amplitude and phase. If the cell impedance at the applied voltage and frequency is primarily resistive, the phase angle between the voltage and current should be nominally zero. Phase angles significantly more positive than zero indicate onset of potentiostat speed limitations or inductive behavior in the cell. Phase angles significantly below zero are characteristic of capacitive, kinetic, or diffusional control of the cell impedance.

Ideally,  $R_u$  should be measured at the highest frequency at which the phase angle of the cell impedance is zero degrees. At this frequency,  $R_u$  is the "real" portion of the cell impedance.

Note that many systems have a quite low  $R_u$  value, indicating that  $iR$  compensation is not required. The system shown in the figure above had a measured  $R_u$  of about 8 Ohms. A typical CV current in this system was  $10\ \mu\text{A}$ . At this current, the error in the applied voltage in the absence of  $iR$  compensation is  $80\ \mu\text{V}$ . In general,  $iR$  errors that are less than  $1\ \text{mV}$  should be ignored, as in this case.

The range and resolution of the positive feedback  $iR$  circuit is limited. Assume for the moment that positive feedback  $iR$  compensation is needed for a cyclic voltammetry (CV) experiment. CV is normally done using a fixed current measurement range. There is an equivalent current measurement resistance associated with each range. This resistance, known as  $R_m$ , can be calculated as  $3V/\text{Current Range}$ . For example, on the  $300\ \mu\text{A}$  range,  $R_m$  is  $10\ \text{k}\Omega$ . The maximum  $iR$  correction on any range is the value of  $R_m$ , and the resolution of the setting is  $0.025\% R_m$ . On the  $300\ \mu\text{A}$  range,  $R_u$  can be set between  $2.5\ \Omega$  and  $10\ \text{k}\Omega$ , with a resolution of  $2.5\ \Omega$ .

In general, the user should avoid entering the full measured  $R_u$  value into tests that use positive feedback  $iR$  compensation. A potentiostat is likely to oscillate when fully  $iR$  compensated. We recommend that 75 to 95% of the measured  $R_u$  value be applied as compensation. In our example system, with an  $8\ \Omega$   $R_u$ , the positive feedback value, if entered at all, would therefore be entered as a value between  $6\ \Omega$  (at 80% compensation) or  $7.6\ \Omega$  (at 95% compensation).

## Ru Estimation - Experimental Sequence

1. A Runner window is created by the Framework and the "Get Ru.Exp" script is run in this window.
2. The script creates the Setup dialog box, which becomes the active window and accepts changes in the experimental parameters. This Setup box remembers the experimental settings from the last time this script was run. To restore the parameters to the values defined in the script, select the Default button. If the Advanced Pstat Setup is toggled to the on position a second Setup dialog box contain hardware configuration details will become the active window allowing the user to modify the hardware configuration used during the experiment.
3. The script next obtains the use of the potentiostat specified during Setup. No data file is created in this script, the user should record the final  $R_u$  manually.
4. The following sequence of events occurs to measure the  $R_u$  of the cell:
  - i. The selected voltage is applied to the cell.
  - ii. A controlled potential EIS measurement is made at a  $7\ \text{mV}$  rms applied amplitude and a  $100\ \text{kHz}$  frequency.
  - iii. The phase angle and real portion of the impedance are recorded.
  - iv. The frequency is successively decreased to 500, 200, 10, 5, 2, 1,  $0.5\ \text{kHz}$ . An EIS measurement is made at each frequency.
  - v. If the phase angle is less than  $\pm 7.5^\circ$  at the frequency in use, the "real" impedance is reported on the computer screen as the  $R_u$  estimate.
  - vi. If the phase angle is more than  $7.5^\circ$  the phase angle between the last two EIS measurements is compared.
  - vii. If the phase angle diverges from zero at lower frequency, the real impedance at the higher frequency is reported as  $R_u$  and the user is warned that the  $R_u$  determination may be in error.
5. Once an  $R_u$  value has been estimated the cell is turned off. The measured  $R_u$  value is displayed in a dialogue box. The script then waits for you to select Skip. Once you do so, the script closes everything that's open, including the Runner window.

## Ru Estimation - Setup Parameters

Click on each of the setup parameters for a description of its use and function.



## Explain Experimental Control Language

All Gamry Instruments experimental techniques are implemented as user accessible scripts. A script is a specialized computer program used to define and control the sequence of events during an experiment. The language used for these scripts is called Explain™. An Explain compiler is built into the Gamry Framework.

Explain is a simple but powerful computer language designed specifically for writing experimental scripts. Explain was developed from a version of the Smalltalk language, written by David Betz, called Extalk. Extalk is too cumbersome for the needs of experimental control, so we redesigned and simplified it. We feel Explain provides a very crisp, readable, and modifiable script language which preserves most of Extalk's modern object oriented characteristics.

We have given you access to Explain so that you can modify our standard scripts or create scripts of your own. This help section gives you the fundamental information necessary to understand what happens in an Explain script. Explain library functions for each Gamry Instruments application are described in that application's Help Section.

## The Flavor of Explain

Explain is a procedural language based on function calls and sequential execution of statements - similar to FORTRAN, Pascal and C. It has variables where values can be stored. Explain, because of its object oriented Smalltalk heritage, also includes "objects" and "classes". What you won't see in an Explain script are a lot of esoteric characters and extraneous words. We tried to keep Explain's syntax simple. We've also made data declarations unnecessary. If you're familiar with any common computer language, Explain scripts should be easy to comprehend.

Explain also has a built in function library. Some of these functions are general purpose while others are designed for the needs of corrosion measurement. Some of the functions are very simple, while others have several parameters and are quite complex.

## The Mechanics of Explain

An Explain script is a simple ASCII text file you could create with any editor, even the infamous EDLIN. We recommend you use Windows Notepad or the editor window built into the Framework for editing scripts. You cannot use a word processor like WordPerfect® or Word for Windows®. They leave formatting and font information in the script.



Explain uses indentation via tabs to mark program blocks. Many editors, including the DOS EDIT program, substitute spaces for tabs in a file. The Explain compiler is unable to read a file with spaces in place of tabs. Do not use the DOS EDIT program to modify an Explain script!

You run an Explain script by selecting it from the Framework **Experiment** menu. The Framework creates a runner Window as the first step in executing the script. Behind the runner window, a compiler built into the Framework turns the script into a tokenized (machine executable) form. This accounts for the short delay you may notice at the start of each experiment. Following the compilation, an interpreter takes the tokenized code and starts running it. This is when visible changes happen to the runner window. We will refer to these separate phases of running as the Compiler and the Interpreter.

## A Sample Explain Script

The following code is typical of a customized script. It is shown here so that you can get a feel for the overall structure and syntax of a script. If you are not familiar with object oriented programming concepts this code can look particularly frightening. Don't panic, we don't expect you to understand this script yet!

The sample script automates a common corrosion test that determines the critical temperature at which pitting corrosion starts to occur. It does the test by running repetitive scans at different temperatures.

```
include "explain5.exp" ; read in standard Framework library

global Pstat
global Output

function Main()
  Pstat = PSTAT.New("PSTAT", "Pstat0") ; Create Pstat object
  if (Pstat.Open() eq FALSE) ; Open Pstat object
    Warning("Can't open the potentiostat")
    return

; Open the output file. All results will go to this file & used later in analysis.
Filename = "TPSCAN.DTA" ; Assign Filename
Output = OUTPUT.New("OUTPUT", Filename, "Output"); Create Output Object
if (Output.Open() eq FALSE)
  Warning("Can't open the output file ", Filename)
  return

Tinit = QUANT.New("TINIT", 25.0, "Start Temp. (C)") ; Initial Temp Object
Tfinal = QUANT.New("TFINAL", 50.0, "Final Temp. (C)"); Final Temp Object
Tstep = QUANT.New("TSTEP", 5.0, "Temp. Step (C)") ; Temp Step Object

; Ask user to fill in values for Tinit, Tfinal, Tstep
Setup("Critical Pitting Temperature Scans", Tinit, Tfinal, Tstep)

Printl("Pitting Temperature Tests")
Tinit.Printl() ; Print the Temperature Setup Parameters to output file
Tfinal.Printl()
Tstep.Printl()

; Create a cyclic ramp generator
; running from V=-1.0 V to 2.0 Volts to 0 Volts at 10mv/sec
; sample at 1 sec intervals
Signal = VUPDN.New("SIGNAL", Pstat, -1.0, 2.0,0.0, 0.010, 0.010 1.0)
Pstat.SetSignal(Signal) ; Specify the signal for the Pstat object

T = Tinit.Value() ; Extract values from objects
Tf = Tfinal.Value() ; and store in variables
Ts = Abs(Tstep.Value())
if (T lt Tf) ; Temp scan going up?
  while (T lt Tf)
    SetTemp(T)
    if (PitScan( -1.0) eq FALSE) ; record a scan
      break
  T = T + Ts
else ; T > Tf temp scan going down
  while (T gt Tf)
    SetTemp(T)
    if (PitScan( -1.0) eq FALSE)
      break
  T = T - Ts

SetTemp(NIL) ; Clean up
Output.Close()
Pstat.Close()
return

function PitScan (Vinit)
; Create a curve to be run.
Curve = CPIV.New("CURVE", Pstat)

Pstat.SetStability(StabilityNorm) ; Set the potentiostat I/E Stability
Pstat.SetIchFilter(IchFilterSlow) ; Set the A/D I filter
Pstat.SetCASpeed(CASpeedNorm) ; Set the Control Amp Stability
```

```

Pstat.SetIERange(0.1) ; Set the IE converter for Big currents
Pstat.InitSignal() ; Initialize the Signal
Pstat.SetCell(CellOn) ; Turn on the cell

Pstat.FindIERange() ; Run a current autorange scan
; so the initial points will be in the correct range
Status = Curve.Run() ; Actually run the curve here
Pstat.SetCell(CellOff) ; Turn off the cell
Curve.Printl() ; output curve results

Status ; make status the last thing evaluated
return ; so it will be then returned value

function SetTemp (T)
if (T eq NIL)
SetTemp(25.0) ; Recursive Function call!
return
Pstat.SetAnalogOut((T-25.0)*20.0) ; Controller = 20 mV/C, To = 25C

```

## Functions & Control Flow

Functions in Explain are very similar to functions in C or Pascal. As in most modern functional languages, each function has a name, an argument list, and a list of statements to be executed in sequence. Explain, like common procedural languages, uses functions to manage execution flow.

```

function SwapPrint(A, B)
if (A lt B)
;Exchange A & B
Temp = A
A = B
B = Temp
Count = Show(A, B)
return

function Show(V1, V2)
Printl("1st Value = ",V1)
Printl("2nd Value = ",V2)
V1+V2 ; last evaluated value is returned
return

```

Note the following Explain features:

- There can be function arguments (e.g. 'A', 'B', 'V1', 'V2').
- There can be local variables within a function (e.g. 'Temp').
- There are no BEGIN or END statements. Explain uses indentation (tabs) to indicate blocks of statements.
- There is no end of statement marker.
- Explain is case sensitive. Keywords are in lowercase.
- Comments begin with the semicolon ";" character and end at the end of the line.

Program statements are generally one line long but they can be continued over several lines. When statements are continued, they must have the same indentation level as the first line of the statement. For example, Function1 below has three parameters:

```
Function1("Parameter1 is very long title string which may not be shown completely if you do
not scroll to the right and look at it.",param2, param3)
```

This can make the code difficult to read. You can force the compiler to disregard indentation changes by putting an ampersand "&" as the first character on a line, for example:

```
& Function1("Parameter1 is still a very long title string but will be shown
& completely because we used a continue symbol.", param2, param3)
```

As you can see, the second code segment is easier to understand.

## A Short Aside about Data types

Experienced programmers may be wondering about the variables, 'V1', and 'Temp', used in the [Functions and Control Flow Section](#). What data type are they? Are they bytes, integers, long integers, reals or double length reals? In Pascal, you would declare the function as:

```
function Show(V1, V2 : integer) : integer
```

whereas in Explain it is

```
function Show(V1, V2)
```

In Explain, the type accompanies the piece of incoming data! In this example, the variable 'V1' in Show will acquire the type and value of 'A' in SwapPrint. Also when the statement 'Temp = A' is executed, the variable 'Temp' will acquire the type and value of 'A'. This lack of data typing for function arguments comes from Explain's Smalltalk heritage.

## Library Functions

Not all functions are Explain language functions contained within the script itself. Explain also includes library functions written by Gamry Instruments in compiled C. To access a compiled library function we use the "callin" keyword followed by a string identifying the compiled function by name. For example:

```
function Printl callin "Printl"
```

The function, Printl is used just like an interpreted function such as SwapPrint shown in the [Functions and Control Flow](#) section.

## Assignments

In programming terminology, an assignment is a statement that attaches a value to a variable. Assignments use the common syntax:

```
Temp = "abc"
```

In Explain an assignment transfers both the value and the type of data. In this example, the variable 'Temp' becomes a STRING with the value "abc" (quotes not included). You can make assignments from one variable to another:

```
A = 5
Temp = A
```

After the first assignment variable 'A' has the data type INDEX and the value, 5. (INDEX is a signed integer but we'll define it more exactly in the [Data Types section](#)). After the second assignment, Temp also has the data type INDEX and the value 5.

## if...else

You can control execution of alternative blocks of statements using the syntax:

```
if (Test)
  Statement1
else
  Statement2
```

The Test is an expression which evaluates to a BOOLEAN value, either TRUE or FALSE. In most cases, Test is a comparison such as (x lt 256).

An 'if' can control a block of statements. Indenting shows the scope of the block:

```
if (Test)
  Statement1
  Statement2
  Statement3
  Statement4
```

Statements 1, 2, & 3 are under the control of the test. If the test is TRUE, all the statements are executed. If it is FALSE, only Statement4 is executed.

You can extend the syntax using the `'else'` keyword:

```
if (Test1)
  Statement1
else if (Test2)
  Statement2
else
  Statement3
```

## Loops

Looping expressions come in three flavors, `'while'`, `'repeat'`, and `'loop'`. The `'while'` statement syntax looks like:

```
while (a < b)
  a = a + 1
  b = b - 1
... ; next statement after the loop
```

This loop starts by evaluating the test expression, `a < b`. If the test evaluates as `TRUE`, the indented statements are executed in order. After the last indented statement is executed, the test is repeated. As long as the test is `TRUE`, the loop is repeated.

The repeat loop is similar:

```
repeat
  Statement1
  Statement2
  Statement3
until (Test)
... ; continuing after loop
```

This works just like you might imagine. The statements will be repeated over and over again until the test is `TRUE`. The statements in the `'repeat'` loop are always executed at least once, unlike those in the `'while'` loop which can be skipped completely.

The last loop is even simpler:

```
loop
  Statement1
  Statement2
  Statement3
... ; continuing after loop
```

If you see a problem with this syntax, you're correct -- `'loop'` will repeat ad nauseam. To escape the endless loop, use the `'break'` statement:

```
loop
  a = a - 1
  if (a < 0) break
  print(a)
... ; continuing after loop
```

The `'break'` causes the loop to terminate without executing the rest of the statements. This syntax allows you to create any type of loop structure.

There is also a similar `'continue'` statement which skips the rest of the statements but doesn't terminate the loop:

```
loop
  a = a + 1
  if (IsPrime(a))
    continue
  PrintFactors(a)
  if (a > 170)
    break
... ; continuing after loop
```

The `'break'` and `'continue'` expressions also work with the `'while'` and `'repeat...until'` loops.

## Return Values

A function returns the value and type of the last expression evaluated. Functions will always have a return value, although it may not be meaningful. The `'return'` statement can be used to terminate the function before its last statement, e.g.:

```
if ( A lt B )
  A
  return
A = A * B
```

If the test is true, the type and value of A is returned immediately. The next statement is never executed. Note that we don't use the C syntax, `'return A'`.

Assignments can be made from the results of function evaluations:

```
Count = Show(A,B)
```

Whatever 'Show' returns, both type and value, become the type and value of 'Count'.

## Included Files

You can hide commonly used items in a separate ".EXP" file. It can be included in a script by the statement:

```
include "filename.exp"
```

When this statement is read, the compiler closes the original file and then opens "filename.exp" and begins reading it. The statements in "filename.exp" are treated no differently than statements in the original file. When the compiler is finished with "filename.exp", it reopens the original file and resumes reading it from the statement following the `'include'` statement. This is similar to the C language `#include`.

All of the Framework `'callin'` library functions and class definitions are hidden in files included by "explain5.exp" which is included in each standard script. You should also include this file in each script you write. Use the line:

```
include "explain5.exp"
```

Without this line you won't have access to the Framework library functions.

## Operators

The operators fall into the following groups:

Boolean: A `or` B (logical inclusive or)

A `and` B (logical and)

A `xor` B (logical exclusive or)

Equality: A `eq` B (equal)

A `ne` B (not equal)

Comparison: A `gt` B (greater than)

A `lt` B (less than)

A `ge` B (greater than or equal to)

A `le` B (less than or equal to)

Shift: A `>>` B (shift right by B bits)

A `<<` B (shift left by B bits)

Sum: A `+` B (add)

A `-` B (subtract)

Mult: A `*` B (multiply)

A `/` B (divide or modulo)

A `%` B (remainder)

Unary: `not` A (logical not)

`-`A (negate)



```
+A (identity)
A (identity)
```

Note that we have included the identity operator for completeness.

## Expressions

In programming terminology, an expression is a statement, or portion of a statement, that results in a single value. An example of an expression is '2+3', which evaluates to 5. When the Explain interpreter gets to an expression, it evaluates it term by term to arrive at the expression's value.

Expressions can be combined, e.g.:

```
A + B * C
A eq B or C eq D
```

Assignments and tests can be made with combined operations:

```
D = A + B * C
if (A eq B or C eq D)
  Statement
```

## Precedence

As you go down the [list of operators](#), the operators bind more tightly to their arguments. "Binding" tells the compiler which expressions to evaluate first in a complex statement. Tightly bound operators are evaluated before more loosely bound operators. Practically this means the expressions parse as:

```
A + B * C --> A + (B * C)
```

```
A eq B or C eq D --> (A eq B) or (C eq D)
```

Expressions bind tighter toward the left, e.g.:

```
A / B * C --> (A / B) * C
```

You can use parentheses to force other binding, e.g.:

```
(A + B) * C
```

```
A / (B * C)
```

Again if you've programmed in C, Pascal, or Fortran, this should look familiar.

## Variables

Explain allows you to store data in variables. Whenever you have variables (and what computer language would be useful without them?), you need to be aware of three issues:

```
Data Storage -- Where data is stored and when is it accessible.
Datatype -- What format the data is in, and what can be done to it.
Data Value -- The actual value(s) a piece of data represents.
```

For example, consider the simple Explain assignment statement:

```
A = 1
```

The 'A' refers to a variable in which we've stored a value of 1. Later on we can use the data in 'A':

```
B = A
```

Now 'A' and 'B' both contain the value 1.

## Data Storage

There are three data storage categories for variables;

 [Locals](#)

 [Globals](#)

 [Arguments](#)

## Local Variables

Local variables are created "on the fly" while a function is being executed. They are only known to the function that creates them. For example, the following code contains an error:

```
function ExplainDelay()
  Now = Time()      ; Record the time in local variable Now
  WaitUntil(30.0)   ; Wait for 30 seconds
  return

function WaitUntil(RequestElapsed)
  repeat
    Elapsed = Time()-Now ; Calculate elapsed time (ERROR!)
  until (Elapsed > RequestElapsed)
  return
```

The error is a consequence of the variable 'Now' being a local variable in function ExplainDelay. It is only known while the interpreter is executing function ExplainDelay. The variable 'Now' in function WaitUntil is a different variable known only to function WaitUntil. WaitUntil's 'Now' is not the same as ExplainDelay's 'Now'!

Local variables can appear anywhere in a function. They do not have to be predeclared as in Pascal. They are created when they are first referenced (usually by writing to the variable). Take the example above written (correctly) as one function.

```
function ExplainDelay()
  Now = Time()      ; Record current time in Now
  repeat
    Elapsed = Time()-Now ; Calculate elapsed time in Elapsed
  until (Elapsed > 30.0)
  return
```

Note the variable 'Now' is created in line 2 and the variable 'Elapsed' is created in line 4. Explain has the advantage of not requiring a list of data declarations before the action of a function begins. There is a price to pay, however. Consider the incorrect function below:

```
function Snooze()
  Now = Time()
  while (Elapsed>30.0) ; Test for elapsed time passed (ERROR!)
    Elapsed = Time()-Now
  return
```

In this example, the variable 'Elapsed' is read before it is set! Explain will just create the variable, 'Elapsed' and give it a special value of NIL. The Explain compiler will not complain about this syntax. The interpreter, however, will complain that you are trying to compare a NIL value to a REAL value.

There is another problem with local variables. When you leave the function where they are declared, their values disappear! You can not permanently store a value in a local variable. For example, consider the following incorrect code:

```
function SetSpeed(NewValue)
  if (NewValue ne NIL)
    RepRate = NewValue
  else
    RepRate = RepRate+1 ; ERROR!
  return

function Main()
  SetSpeed(1)
```

```

while (RunExperiment() eq 0) ; return of zero = experiment was ok
  SetSpeed(NIL) ; go faster by incrementing RepRate
. . . ; continue after loop

```

The problem is that local variable 'RepRate' doesn't live past the return statement in function SetSpeed. The first time SetSpeed is called using parameter 1, 'RepRate' is set correctly. However, by the time SetSpeed is called with value NIL, the original 'RepRate' has disappeared and the new 'RepRate' has an incorrect value.

## Global Variables

To get around the Local variable problem, it is possible to create a Global variable. Global variables are declared outside a function and are visible throughout the entire script. Global variables are declared outside a function using the keyword `'global'`, for example:

```
global GVar
```

or

```
global GVar = 5
```

Using global variables, you can keep status information past the end of a function. Making 'RepRate' a global variable fixes the problem described under [Local Variables](#):

```

global RepRate

function SetSpeed(NewValue)
  if (NewValue ne NIL)
    RepRate = NewValue
  else
    RepRate = RepRate+1
  return

function Main()
  SetSpeed(1)
  while (RunExperiment() eq 0) ; return of zero = experiment was ok
    SetSpeed(NIL) ; go faster by incrementing RepRate
  . . . ; continue after loop

```

## Function Argument Variables

There is one other type of variable in Explain's repertoire - the function argument. You've already seen arguments in the examples in the [Global](#) and [Local](#) Variables discussions, but to reiterate:

```

function Main()
  ExpSpeed = 1.5
  SetSpeed(ExpSpeed)
  SetSpeed(1.5)
  SetSpeed(NIL)

function SetSpeed(NewValue)
  ... ; function's code
  return

```

The variable, 'NewValue', in function SetSpeed() is an argument. It behaves a lot like a local variable in that it is known and reliable only while the interpreter is executing the statements inside the SetSpeed() function. If you call SetSpeed() repeatedly, 'NewValue' will not be remembered between calls.

There is another issue to consider when you use arguments. Professional programmers call this the "Call by Value versus Call by Reference" issue. Explain uses a Call by Value for simple data types. This means that the caller function gives the called function a copy of the data value rather than the data itself. It is easier to see this with an example:

```

function Main()
  Speed = 1.5
  printl("Main: Speed = ",Speed)
  Modify(Speed)
  printl("Main: Speed = ",Speed)

```

```

return

function Modify(Speed)
  println("Modify: Speed = ", Speed)
  Speed = Speed + 1.0
  println("Modify: Speed = ", Speed)
return

```

The output from this program will look like:

```

Main: Speed = 1.5
Modify: Speed = 1.5
Modify: Speed = 2.5
Main: Speed = 1.5

```

So the argument 'Speed' in function Modify() is modified, but the original value in [function Main\(\)](#) is left alone. Even though they have the same name and origin, they are two completely different values. Again, you can get around this effect by using global variables.

More complex data objects are implemented as Call by Reference. We'll show you an example in the [Object Scope and Lifetime](#) section.

## Data types

Explain has fundamental data types which describe simple values. It also has more complicated data types such as VECTORS and CLASSES which will be described later. Explain has the following fundamental data types:

 [INDEX](#)

 [REAL](#)

 [BITS](#)

 [BOOL](#)

 [STRING](#)

 [NIL](#)

The fundamental types can be directly manipulated as in most high level languages. The concept of data type encapsulates several different issues:

- How is data stored?
- What are the allowed values?  
BOOL = TRUE or FALSE
- How can values be retrieved or modified?, e.g.  
A = A + 1  
A.SetValue(3.0, TRUE)
- How are constants represented in the script?, e.g.  
TRUE (BOOL Constant)  
5 (INDEX Constant)  
17.3E5 (REAL Constant)
- What operations can be performed on the data?, e.g.  
1 - 3  
TRUE `eq` FALSE  
0xFFFFFFFF `xor` 0xAF143
- Where can data of a given type be used?  
Sleep("Gamry") ; *illegal since Sleep() needs INDEX*

Data types are defined for constants, variables, the results of expressions (e.g. A eq B has type BOOL), and the return values of function calls. Parameter usage will be described with each individual function.

## Value/Type Binding

Explain data are stored as variables similar to those in common programming languages like Pascal or FORTRAN. However, the data type and the data value are chained together. In Pascal and C, the data type is bound to the variable. So when you declare:

```
float Var1; /* C Language */
Var1 = 5.0;
... /* other statements */
Var1 = 6;
```

Var1 is a 'float' for ever and always. In C, when the value '6' is assigned to Var1, it is first converted to the float value 6.0 and then stored in Var1.

In Explain, you can have a similar set of statements:

```
Var1 = 5.0 ; Explain
... ; other statements
Var1 = 6
```

After the first statement 'Var1' has the data type REAL and the value 5.0. After the second statement 'Var1' has the data type INDEX and the value 6! So variables do not always have the same data type.

## Mixed Mode Operations

There will be cases in which you need to perform operations with data of different types. In Explain it is legal to perform mixed mode comparisons and arithmetic operations that involve both a REAL and an INDEX quantity. In all mixed mode operations, the INDEX is "promoted" to a REAL prior to the operation.

REAL Sum INDEX -> REAL (1.2 + 2-> 3.2)

REAL Mult INDEX -> REAL (1.2 / 2 -> 0.6)

REAL Equality INDEX -> BOOL (1.01 **eq** 1 -> FALSE)

REAL Comparison INDEX -> BOOL (1.01 **gt** 1 -> TRUE)

Other mixed mode operations are not recommended.

The Explain library includes some data type cast functions that allow you to force conversion of one data type into another. Examples include the ROUND() and INDEX() functions.

## Classes and Objects

Some of the most useful features of Explain are its classes. An Explain class is a complex data type. The class describes how the data is stored, accessed and modified. It is therefore more than just a storage location for the data. An object is a particular instance of a class. For example, Joe and Al are both GUYS. Joe and Al are two objects, GUYS is a class.

Classes and Objects are features that Explain inherited from its Smalltalk grandparent. If you've never seen an object-oriented language, the concepts may seem strange. In traditional languages data doesn't know how to modify or print itself! Once the concepts are mastered, most programmers find them easy to use and find they provide real advantages.

Let's look at a quick example before getting into the details. Real experiments are a little more complicated than the one shown below.

```
class CURVE
  cfunction New callin "CurveNew"
  ifunction Run callin "CurveRun"
  ifunction Printl callin "CurvePrintl"

class VRAMP
  cfunction New callin "VrampNew"
  license Signal callin "RampSignal"

function Main()
  ; Create a ramp generator
```

```

Sig = VRAMP.New("SIGNAL", Pstat, 1.0, 5.0, 0.010, 1.0)

; Specify the Signal for use with the Pstat object
Pstat.SetSignal(Signal)

; Create a curve to be run.
Crv = CURVE.New("CURVE", Pstat)

; Run the curve
Status = Crv.Run()

; Print the results to the output file
Crv.Printl()

```

Refer back to this script as we describe [Classes](#) and [Objects](#).

## Classes

Think of a class as a definition. It defines how objects of that class behave.

In the [sample script](#), there are two classes, a CURVE class, and a VRAMP class. CURVE defines data acquisition and display objects. When an experiment is run, data is stored in a CURVE object. A CURVE object displays itself on the screen as a real time plot. A VRAMP object is a signal generator that scans applied voltage between two points.

Each class has a list of functions which can be applied either to the class or to objects created from the class. You won't see the actual data layout as you would in a C language structure or Pascal Record. That information is hidden. You can only see the various functions used to create or access the object. You must rely on these functions to either modify an object or access values within it.

There are three types of functions contained in a class: Class Functions, Instance Functions and Licenses. These are indicated by the keywords "[cfunction](#)", "[ifunction](#)", and "[license](#)", respectively. Each of these is discussed in the following help sections.

## Objects

The actual item which is created and manipulated is an object. Each object is created as an instance of a specific class. In Explain, we don't have objects defined by more than one class. Objects have areas of storage assigned to them. They have a specific lifetime which we'll describe in the following help sections.

## Class Functions (cfunction)

A class function is a function that is applied to the class definition itself. It is most often used to create an object of the class. For example:

```
Sig = VRAMP.New("SIGNAL", Pstat, 1.0, 5.0, 0.010, 1.0)
```

'New' is a [cfunction](#) of [class](#) VRAMP. It is applied directly to the class name using syntax: VRAMP.New. The function New creates a new object of [class](#) VRAMP. This object is assigned to the local variable Sig. The actual interpretation of the parameters following the function is up to each class.

```

VRAMP.New
Parameter 1: Tag -- STRING -- text used when object is printed or stored
Parameter 2: Pstat -- PSTAT -- Valid Pstat object
Parameter 3: Einit -- REAL -- the initial voltage
Parameter 4: Efinal -- REAL -- the final voltage
Parameter 5: Estep -- REAL -- the voltage step size
Parameter 6: Tstep -- REAL -- the time between steps
Return value object of type RAMP

```

A few classes have cfunctions besides New. These functions can affect all objects of that class. See the [class definition](#) for details about each class.

## Instance Functions (ifunction)

Instance functions are the main tool for modifying or extracting data stored in an object. Some object oriented languages call these things "messages". But since it behaves like a function, we've called it a function.

Instance functions are applied to objects rather than classes. In the sample script:

```
Status = Crv.Run()
```

Crv is an object of [class](#) CURVE. Therefore [ifunction](#) Run() can be applied to it. There may be other ifunctions with the name Run() in other classes. However, Explain will make sure the correct Run() is used.

Instance functions may have parameters and may return values (such as Status in the above example). Again, check the [class definition](#).

## Licenses

The license is a novel concept in Explain. A license encapsulates a whole group of abilities or skills. Different groups of skills have different license names. An object may have more than one license. We use this metaphor in real life all the time. A plumber, for example, has a plumber's license and a driver's license. Each license attributes a whole group of skills to the plumber. Each object that has a given license may implement the individual skills differently.

To see how we implement this in Explain look at his portion of the example script:

```
class "RampSignal"
...   callin Signal license VRAMP
; other initialization statements
Sig = VRAMP.New(...)
Pstat.SetSignal(Sig)
Crv = CURVE.New("CURVE", Pstat)
```

Objects of the [class](#) VRAMP have the Signal [license](#). Other classes may also have the Signal [license](#). One example is the VSTEP class, which is used to generate a step signal rather than a ramp signal. A PSTAT object needs an object with the Signal license to determine the applied voltage at each point and to define the number of points in the curve. After the PSTAT object is created, the name of a Signal object is passed to the Pstat using the Pstat.SetSignal function.

## Object Scope and Lifetime

Objects exist only as long as they are referenced. When the class function New is called, it assigns the new object to a variable name. The same object can then be assigned to other variables. For example, this code fragment creates a simple object and then assigns it to a second variable.

```
X = QUANT.New("SampTime", 1.0, "Sample Time (sec)")
Y = X
... ; other statements
X = 42
```

In this example, both X and Y initially reference the same object. This object continues to exist and take up space in memory until [all](#) references to the object have been reassigned. Thus at the end of the fragment, the object still exists and can be accessed by ifunctions of object Y even though its original variable X has been reassigned.

Once all references to the object have been reassigned, the object is "deleted" and memory used for the object is free to be used by other objects and variables. This may not be important for simple variables such as the QUANT in the example, but can be critical for memory intensive objects such as data curves.

Often the reassignment that deletes an object occurs naturally in the program flow. Assigning a new value to the variable reassigns the variable. If you want to explicitly delete an object, you can do so by assigning all active references to the object to the data type NIL. In the example above, adding the line

```
Y = NIL
```

freed up the memory assigned to the QUANT type object.

The scope and lifetime of objects created and assigned within functions also needs to be discussed. Look at the following function:

```
function Create&Use()
var1 = CURVE.New("Curve1", ...)
Use(var1)
var2 = 42
return
```

In this example, 'var1' is a local variable. It has the value of the CURVE object just created and the data type CURVE. We can refer to this

object by its tag, var1. When 'var1' is passed to function Use(), it is equivalent to sending Curve1 to Use(). When the function Create&Use() is exited at the return statement, 'var1' disappears and the CURVE object is deleted.

Notice that a function can also return a reference to an object. In this case the returned object is not deleted as the function is exited. See the next example:

```
X = Create()

function Create()
  var1 = CURVE.New("Curve1", ...)
  ... ; other statements
  var1
  return
```

Remember, the last expression evaluated will be the return value from a function. Since the return value of this function is var1, the CURVE object is not deleted. Be careful, you can inadvertently return a memory intensive object from a function creating unexpected memory usage problems.

In most cases, you can ignore the issue of object lifetime and the related issues of memory usage. The Explain interpreter deletes all objects created by a script when the script's Runner window is closed. However, these issues can become critical if you write large scripts with multiple data curves.

## Vectors

A VECTOR is another complex Explain data type. It is analogous to the arrays that are common in other computer languages. Like an array, a VECTOR contains a number of elements that are accessed by a numerical index into the VECTOR. Unlike traditional arrays, an Explain VECTOR can contain a mixture of data types including both simple data types and objects.

The only information needed to create a new VECTOR is the number of elements that will be required. The data type of the VECTOR elements is not specified when the VECTOR is created. For example, this statement creates a new 6 element VECTOR named XVector.

```
XVector = VectorNew(6)
```

The elements of the new VECTOR are all initialized to the NIL data type.

The index (which must be of the INDEX data type) used to access elements of the VECTOR is zero based. The 6 elements of the XVector are therefore numbered 0 to 5. To access a Vector element, you give the VECTOR name followed by the index in square brackets. For example, to assign the REAL value 5.0 to the third element of your XVector, you can use the line:

```
XVector[2] = 5.0 ; third element becomes a REAL with value of 5.0
```

Notice that the Nth VECTOR element has an index of N-1 because a VECTOR's index is zero based.

You can assign the contents of the VECTOR element to a regular Explain variable (in this example called 'Variable2') as follows:

```
Variable2 = XVector[2] ; Variable2 becomes a REAL with value of 5.0
```

The index into a VECTOR will often be a loop counter. For example, this code fragment stores the square of the integers 0 to 9 in the elements of a VECTOR called Squares.

```
Squares = VectorNew(10)
i = 0
while ( i lt 10)
  Squares[i] = i * i
  i = i + 1
... ; next statement after the loop
```

Similar to Explain variables, the elements take the data type of data that is assigned to them. This line reassigns the third element of XVector to an object of the CPIV curve class.

```
XVector[2] = CPIV.New(...) ; New() function arguments left out for clarity
```

You can still access all the Cpivot object's functions. For example, objects of the CPIV class have an instance function Printl() that prints the curve to the output file. To call this function using the Cpivot object in XVector use this line:

```
XVector[2].Printl()
```



A VECTOR element can contain another VECTOR. You can therefore generate a construct similar to a 2 dimensional array:

```
XYVector = VectorNew(5)

i = 0
while ( i lt 5)
  XYVector[i] = VectorNew[10]
  i = i +1
... ; next statement after the loop
```

This code generates a 2 dimensional VECTOR with 5 columns and 10 rows. To access the Nth element in row M, the syntax is XYVector[N-1][M-1]. Again, remember that VECTOR indexes are zero based.

## Gamry Framework: Explain Library

The Gamry Framework comes with a built-in set of general purpose library functions and classes. The Framework library contains both compiled C and Explain routines. These library routines provide essential services such as I/O, time keeping, potentiostat control and curve acquisition.

Each Gamry Instruments application also uses library functions and classes that are specifically designed for that application's measurements. These items are discussed in the Operator's Manuals for each application.

We will use UPPERCASE names to refer to class names and Mixed Case names to refer to object names:

"LABEL" refers to a class.

"Label" refers to an object.

Since Explain is case sensitive, LABEL and Label are two different names. In this manual, when a MixedCase object name is the same as a UPPERCASE class name, you may assume the object is of that class.

When an ASCII tab character appears in sample output, it will be indicated as:

TEXT <tab> MORE TEXT

The terms CURVE and Curve object are used in this section to refer to the various classes of objects that acquire and hold run time data. An example is the OCV class.

## Gamry Framework: Library Routines

### Math functions

[Abs\(\)](#) Calculate the absolute value of a number  
[Exp\(\)](#) Calculate ex of a number  
[Index\(\)](#) Convert a quantity to an INDEX  
[LineOpt\(\)](#) Find a line noise rejecting frequency  
[Log\(\)](#) Calculate natural logarithm of a number  
[Log10\(\)](#) Calculate base-10 log of a number  
[Modulus](#) Convert Real, Imaginary to Modulus  
[Phase](#) Convert Real, Imaginary to Phase  
[Pow\(\)](#) Calculate number raised to a power  
[Rand\(\)](#) Generate a random number  
[Real\(\)](#) Convert quantity to floating point  
[Round\(\)](#) Round off a REAL to N decimal places  
[Sqrt\(\)](#) Calculate square root of a number

### Trigonometry Functions

[ArcCos\(\)](#) Arc Cosine  
[ArcSin\(\)](#) Arc Sine  
[ArcTan\(\)](#) Arc Tangent  
[Cos\(\)](#) Cosine  
[Cosh\(\)](#) Hyperbolic Cosine  
[DtoR\(\)](#) Convert Degrees to Radians  
[NormalD\(\)](#) Normalize an angle to +/- 360 Degrees  
[NormalR\(\)](#) Normalize an angle to +/- 2 $\pi$   
[RtoD\(\)](#) Convert Radians to Degrees  
[Sin\(\)](#) Sine  
[Sinh\(\)](#) Hyperbolic Sine  
[Tan\(\)](#) Tangent

[Tanh\(\)](#) Hyperbolic Tangent

### Mathematical Classes

class [COMPLEX](#) Complex number class

### String Functions

[Ascii\(\)](#) Convert a character to an Ascii number  
[Char\(\)](#) Convert an Ascii number to a character  
[IsAlnum\(\)](#) Check if character is alphanumeric  
[IsAlpha\(\)](#) Check if character is letter  
[IsDigit\(\)](#) Check if character is numeric  
[IsLower\(\)](#) Check if character is lowercase letter  
[IsPunct\(\)](#) Check if character is punctuation  
[IsSpace\(\)](#) Check if character is white-space  
[IsUpper\(\)](#) Check if character is uppercase letter  
[IsXdigit\(\)](#) Check if character is hexadecimal digit  
[StrCmp\(\)](#) Compare two strings  
[StrGet\(\)](#) Get ith character in a string  
[StrLen\(\)](#) Calculate length of string  
[StrLwr\(\)](#) Convert string to lowercase  
[StrSet\(\)](#) Set ith character in a string  
[StrUpr\(\)](#) Convert string to uppercase

### Time functions

[DateStamp\(\)](#) Generate a string containing a date  
[Time\(\)](#) Set a variable to the current time  
[TimeStamp\(\)](#) Generate a string containing a time

### Statistical functions

[Mean\(\)](#) Calculate Mean of a data set  
[StatsOne\(\)](#) Calculate Statistics on one data set  
[StatsTwo\(\)](#) Calculate Statistics between two data sets

### Delay and Program Control functions

[Abort\(\)](#) Immediately abort a script  
[Dawdle\(\)](#) Halt script until SKIP selected  
[Execute\(\)](#) Launch a child script  
[ExecWait\(\)](#) Launch a child script and wait for completion  
[Pause\(\)](#) Pause script for a set number of seconds  
[Sleep\(\)](#) Tell a script to sleep until a specified time  
[Suspend\(\)](#) Allows a user to continue or abort  
[WinExec\(\)](#) Start up a Windows application  
[Yield\(\)](#) Let other Windows programs gain control

### Output functions

[Print\(\)](#) Output to the current output file  
[Printl\(\)](#) Output to file with automatic newline  
[Sprint\(\)](#) Output an item to a STRING

### Video Display functions

[Stdout\(\)](#) Sends output to the STDOUT window  
[StdoutActivate\(\)](#) Brings STDOUT window to foreground  
[Error\(\)](#) Report a fatal error and terminate the run  
[Headline\(\)](#) Update the headline above the data curve  
[Notify\(\)](#) Update runner window status display  
[Notify2\(\)](#) Secondary Update runner window status display  
[Query\(\)](#) Ask operator a multiple choice question  
[Setup\(\)](#) Open a dialog box for parameter input  
[Warning\(\)](#) Warn operator - wait for OK to proceed

### Class and Object Manipulation functions

[ClassIndex\(\)](#) Numerical sorting of classes  
[ClassName\(\)](#) Place class name in a STRING  
[ClassNew\(\)](#) Dynamically create a class  
[ClassAddI Sel\(\)](#) Dynamically add an instance variable  
[ClassAddC Sel\(\)](#) Dynamically add a class variable  
[FindClassByName\(\)](#) Return a class given a STRING  
[ObjectNew\(\)](#) Dynamically create a new object

### Setup Disk File functions

[SetupRestore\(\)](#) Recover an old setup on disk

[SetupSave\(\)](#) Save a setup on disk

#### Miscellaneous functions

[Callin\(\)](#) Dynamically acquire DLL library function

[Config\(\)](#) Read an INI file value

[LoadLibrary\(\)](#) Dynamically load a DLL library

[SetConfig\(\)](#) Set an INI file value

[VectorCount\(\)](#) Determine size of a Vector

[VectorNew\(\)](#) Generate a new VECTOR

#### Parameter Classes having Dialog license suitable for Setup

class [CHANNEL](#) Used to setup multiplexed experiments

class [DLGSPACE](#) Used to generate white space in Setup()

class [IQUANT](#) An integer parameter for general use

class [LABEL](#) A short string for general use

class [NOTES](#) A multiline string too long to SAVE/RECALL

class [ONEPARAM](#) Complex class- 1 TOGGLE, 1 QUANT

class [OUTPUT](#) An output file

class [POTEN](#) A potential with vs Eref & vs Eoc info.

class [QUANT](#) A real parameter for general use

class [SELECTOR](#) Radio button selection

class [STATIC](#) String used for descriptive purposes

class [TOGGLE](#) An on/off parameter for general use

class [TWOPARAM](#) Complex class -1 TOGGLE, 2 QUANTS

#### Instrument Driver Classes

class [PSTAT](#) A potentiostat

class [MUX](#) An ECM8 Multiplexer

#### Classes having Signal licenses used by curve functions

class [ICONST](#) A constant current

class [VCONST](#) A constant voltage signal

#### Curve Classes

class [CGEN](#) Generic curve class

class [CURVE](#) A generic name for all this group's classes

class [IVT](#) Generic IVT curve used for conditioning

class [OCV](#) Curve to measure open circuit potential

#### Miscellaneous Classes

class [DATACOL](#) Hold data extracted from CURVE object

class [ARRAY](#) Array class which can hold any object

## Abort()



#### Related Topics

[TYPE:](#) REGULAR FUNCTION

[PURPOSE:](#) Abort a Running Script with no user intervention or prompt.

[USAGE:](#)

Abort()

## Abs()



#### Related Topics

[TYPE:](#) REGULAR FUNCTION

**PURPOSE:** Calculate the absolute value of a number. The number can be an INDEX or a REAL. The value returned is the same type as the argument.

**USAGE:**

Result = Abs(Number)

Number	REAL, INDEX	The number to be absolved
Result	REAL, INDEX	The absolute value of Number

## ArcCos()



*Related Topics*

**TYPE:** REGULAR FUNCTION

**PURPOSE:** Calculate the ArcCosine of a number. The angle will be calculated in radians

**USAGE:**

Result = ArcCos(Number)

Number	REAL	The number whose ArcCos is to be evaluated
Result	REAL	The ArcCos of Number in radians

## ArcSin()



*Related Topics*

**TYPE:** REGULAR FUNCTION

**PURPOSE:** Calculate the ArcSine of a number. The angle will be calculated in radians

**USAGE:**

Result = ArcSine(Number)

Number	REAL	The number whose ArcSine is to be evaluated
Result	REAL	The ArcSine of Number in radians

## ArcTan()



*Related Topics*

**TYPE:** REGULAR FUNCTION

**PURPOSE:** Calculate the ArcTangent of a number. The angle will be calculated in radians

**USAGE:**

Result = ArcTan(Number)

Number	REAL	The number whose ArcTangent is to be evaluated
Result	REAL	The ArcTangent of Number in radians

# ARRAY



**TYPE:** CLASS

Class ARRAY allows the storage of a multidimensional array of data. Elements stored in an ARRAY object can be any valid explain type or object. Each element maintains a type, so one array can hold multiple types of explain objects at one time.

## ARRAY.New()



**TYPE:** CLASS FUNCTION

**PURPOSE:** Create a new object of the ARRAY class. An array can have any number of dimensions provided there is enough available memory to be allocated.

**USAGE:**

```
Array = ARRAY.New(Tag, D1, D2, . . . , Dn)
```

Tag STRING Object Tag used when printing the object

D1 INDEX Index specifying the number of elements in Dimension 1 of the array

D2 INDEX Index specifying the number of elements in Dimension 2 of the array

Dn INDEX Index specifying the number of elements in Dimension n of the array

## Array.Dim()



**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Returns the dimensions of the array. Array.Dim() returns the number of dimensions in the array, while Array.Dim (dimension) returns the size of the specified array column.

**USAGE:**

```
NumDim = Array.Dim()
```

NumDim INDEX The number of dimensions contained by the array

or

```
ColSize = Array.Dim(Column)
```

Column INDEX Valid column of the array

ColSize INDEX Number of elements in the Column

## Array.Get()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Returns the value or object contained in an array element.

USAGE:

Data = Array.Get(x1, x2, . . . , xn)

x1 INDEX First position within the array

x2 INDEX Second position within the array

xn INDEX nth position within the array

Data UNKNOWN Any valid Explain object which was stored in the array

## Array.Set()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Stores a value or object in an array element.

USAGE:

Array.Set(x1, x2, . . . , xn, Data)

x1 INDEX First position within the array

x2 INDEX Second position within the array

xn INDEX nth position within the array

Data UNKNOWN Any valid Explain object to be stored in the array

## Ascii()



### Related Topics

TYPE: REGULAR FUNCTION

PURPOSE: Returns the Ascii number of the first character in a string.

USAGE:

AsciiNumb = Ascii(String)

String STRING String to determine first character.

AsciiNumb INDEX Ascii number of the first character in String.

## Callin()



### Related Topics

**TYPE:** REGULAR FUNCTION

**PURPOSE:** Dynamically assign a callin (compiled) function to a variable. Callin() is equivalent to the static syntax:

```
function Result callin FunctionName
or function Result callin Library @ FunctionName
or function Result callin Library @ FunctionID
```

A result must be assigned to a global variable to be accessible outside the function where Callin is executed.

**USAGE:**

```
Result = Callin(FunctionName)
or
Result = Callin(Library, FunctionName)
or
Result = Callin(Library, FunctionID)
FunctionName STRING Name of the function being located
in a library or Gamry Framework
Library LIBRARY Name of the library to look in
FunctionID INDEX function ordinal number ID
```

## CGEN



### Related Topics

**TYPE:** CLASS

Class CGEN implements a generic curve that you can "stuff" with data directly from Explain. The data in this curve can be plotted on the run-time screen. CGEN curves all have 4 data columns. Unlike the other Explain curve classes, the data columns in a CGEN object do not have preassigned meanings. Column 0 can contain time in one script, current in another, and pH in a third.

A typical lifecycle for a CGEN object is:

```
Cgen = CGEN.New() ; create the object
Cgen.SetAxis(X_AXIS,1,0.5,0,"E(mv)") ; specify linear potential
on X axis
Cgen.SetAxis(1,2,0.001,1,"I(uA)") ; specify log current on
Y axis

i = 0
while (i lt maxpoints)
... ; get point
Cgen.AddPoint() ; add point to curve
Cgen.Activate() ; display plot
Printl("t",Cgen.Sprintl(i)) ; output point to disk
i = i + 1
.... ; rest of code
```

## CGEN.New()



### Related Topics

**TYPE:** CLASS FUNCTION

**PURPOSE:** Create a new object of the CGEN class. The type of real time plot is defined when the object is created. There are two plot types, one for Y versus X data, and one for Y and Y' versus X. The second plot type is used for data with two dependent variables, such as Rp and Eoc versus time plots.

**USAGE:**

```
Cgen = CGEN.New(Tag, PlotView)
```

Tag	STRING	Object Tag used when printing the object
PlotView	INDEX	<i>VIEW_NONE</i> No Curves are displayed <i>VIEW_SINGLE</i> One Curve per plot (as in Potentiodynamic) <i>VIEW_DOUBLE</i> Two Curves per plot (as in Galvanic Corrosion)

*VIEW\_SINGLE* defines 2 axes: *X\_AXIS* and *Y\_AXIS*

*VIEW\_DOUBLE* defines 3 axes: *X\_AXIS*, *Y\_AXIS* (lower y axis), and *Z\_AXIS* (upper y axis)

## Cgen.SprintPoint()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Used in output of data from a CGEN object. A data point from the object is printed to an ASCII string. This string can in turn be printed to the active output file.

The format of the data string is:

```
PointNumber <tab> Data0 <tab> Data1 <tab> Data2 <tab> Data3
```

You can use this function along with the Printl() function to output point i to the output file as follows:

```
Printl("\t",Cgen.SprintPoint(i))
```

**USAGE:**

```
Result = Cgen.SprintPoint(PointNumber)
```

Result STRING The output string

PointNumber INDEX The point to be printed. PointNumber is 0 based. If the requested point has not been taken, a parameter error will be issued.

## Cgen.Activate()



### Related Topics

**TYPE:** INSTANCE FUNCTION



**PURPOSE:** Make this the active curve. It will be displayed if the user has ACTIVE chosen in the CURVE LIST control found on the Runner window. Display of this curve will continue until some other object becomes active.

**USAGE:**

Cgen.Activate()

## Cgen.SetAxis()



**Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Helps to define the real time display of a CGEN object. Associates an axis with a data column in the object. Controls linear/log display formatting and allows labeling of the axis.

The number of axes to be displayed was determined when the object was created via CGEN.New(). The PlotView parameter in CGEN.New() determines whether we have an Y versus X or a Y and Y' versus X plot.

You must call Cgen.SetAxis for each of the axes in the real time plot. For a Y and Y' versus X plot you need 3 calls to Cgen.SetAxis().

**USAGE:**

CGEN.SetAxis(AxisNo, Source, Scale, MinVal, Label, Units)

AxisNo	INDEX	Use the constants below to define which axis is being configured.
		<i>X_AXIS</i> X axis
		<i>Y_AXIS</i> Y axis (Y vs X plot) or Lower Y (Y, Y' vs X)
		<i>Z_AXIS</i> Upper Y axis (Y, Y' vs X only)
Source	INDEX	The column number containing the data to be plotted or use the predefined constants defined in <a href="#">CGEN.DataCol()</a> .
Scale	INDEX	<i>LIN_AXIS</i> The axis is linear
		<i>LOG_AXIS</i> The axis is log
MinVal	REAL	Minimum value used in prevention of "ugly" plot scaling. MinVal is used differently depending on Scale setting.
		On a linear axis, this is the minimum resolution of the plot. The range of values shown on the axis cannot be smaller than MinVal. This prevents excessively fine scaling when plotting data where all the values are identical or very similar.
		On a log axis, this value is substituted for 0's in the data. This prevents infinite log values.
Label	STRING	The label used for the axis. Short names (one or two character labels are preferred).
Units	STRING	The units used for the axis. Short names (one or two character labels are preferred).

## Cgen.Tag()



**Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Return a string containing the Cgen object's tag.

**USAGE:**

Result = Cgen.Tag()

Result STRING The tag associated with the object.

## Cgen.AddPoint()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Place a data point in the Cgen object's data array. The point is added at the next available point in the curve. The data point counter is incremented after the point is added.

Assignment of the significance of each data column is up to the programmer.

**USAGE:**

Cgen.AddPoint(Data0, Data1, Data2, Data3)

Data0 REAL The datum to be placed in column 0

Data1 REAL The datum to be placed in column 1

Data2 REAL The datum to be placed in column 2

Data3 REAL The datum to be placed in column 3

## Cgen.DataCol()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Extract a data column from a CGEN object. This data column is packaged in a DATACOL object that can then be used for further processing.

**USAGE:**

Result = Cgen.DataCol(ColNumber)

Result	DATACOL	A DATACOL object with a tag derived from the Cgen object's tag. The new tag is <CGen Tag>_C<COLNO> where COLNO is the requested column number.
ColNumber	INDEX	Data column, with a range of 0 to 3. The meaning of the data in each column is under user control.

## Cgen.DataValue()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Extract a single data item from a CGEN object. The data item is taken from the ith data point and the jth data column. If no data has been taken the returned value is NIL.

USAGE:

Data Value = Cgen.DataValue(PointNum,ColNumber)

Result	REAL	The returned value
	<i>or</i> NIL	No data acquired yet
PointNum	INDEX	Point number, zero based. If PointNum is out of range, a value from the last or acquired point is returned.
	<i>or</i> NIL	Always return a value from the last acquired point.
ColNumber	INDEX	Data column, with a range of 0 to 3. The meaning of the data in each column is under user control.

## Cgen.Count()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Return the number of points actually acquired in a CGEN object. The number returned is not the number of points allocated.

USAGE:

NumPoints = Cgen.Count()

NumPoints INDEX The number of points in the CGEN object. Zero is returned if no data has been acquired yet.

## CHANNEL



### Related Topics

TYPE: CLASS

PURPOSE: Each channel in a multiplexed technique has a set of parameters that are unique to the test on that channel. A Channel object is a way to encapsulate these parameters and display/edit them on one line in the setup dialog box.

USAGE: The creation and use of Channel objects can be fairly complex. For this reason, you may want to supplement the discussion here with examination of an Explain code script that uses Channel objects. The MUXRPEC.EXP script is a good choice.

There is normally one Channel object created for each multiplexer channel. These Channel objects are normally arranged into an Explain vector. The creation of the Channel objects thus looks something like this:

```
Channel = VectorNew(8)
i=0
while (i lt 8)
  Channel[i] = CHANNEL.New(.....)
  i = i + 1
...
```

If the Channel objects are in a vector, the function call for the Setup() function looks like this:

```
Setup(
& CHANNEL, ; get channel header
& Channel[0], ; 1st row of channel matrix
& Channel[1],
```

etc.

Notice that both the CHANNEL class and Channel objects have a Dialog license. This is a unique feature. A group of Channel objects is displayed in the Setup dialog box as a matrix of editable fields. Each row of the matrix is a channel. Each column is a type of field (test ID, Notes..., etc.).

The CHANNEL class appears in the Setup dialog box as a row of column headers. These headers are intended to label the columns in the channel matrix. With the Setup() call above, the column used to enter electrode area will have the label "AREA" above the column. The other columns will also be labeled according to their function.

The function call syntax for a Channel object that's in a vector looks like this:

```
Channel[1].Ident()
```

## CHANNEL.New()



### Related Topics

TYPE: CLASS FUNCTION

PURPOSE: Create a channel object. The new object contains a number of default variables initially set via this function call. Many of these variables can be changed later on in the Setup() function's dialog box.

USAGE:

```
Channel = CHANNEL.New(Tag, ChannelNo, Active, Ident, System, Area)
```

Channel CHANNEL The new object. Often a VECTOR element.

Tag STRING A short string used to identify the object.  
Printed when the object is printed.

ChannelNo INDEX Allowed range is 0 to 7. The Channel number is used to form the prompt field in Setup() and the title of the notes dialog box.

Active BOOL A flag that indicates whether a channel is to be used or skipped. Can be changed in Setup().

Ident STRING Test identifier for this channel. Corresponds to the Identifier field of non multiplexed experiments.  
Can be changed in Setup().

System STRING Name of chemical system. This string is used to find chemical parameters (Density, Equiv.Wt, Betas) in the SYSTEM.SET database.  
Can be changed in Setup().

Area REAL Electrode area. Can be changed in Setup().

## Channel.SetActive()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Allows a script to set the Active Cell flag in a channel object.

USAGE:

Channel.SetActive(Switch)

Switch BOOL TRUE if channel will be run.  
FALSE if channel to be skipped.

## Channel.Active()



TYPE: INSTANCE FUNCTION

PURPOSE: Reads the Active Cell flag from a channel object.

USAGE:

Flag = Channel.Active()

Flag BOOL TRUE if a channel is to be used.  
FALSE if it is to be skipped

## Channel.Ident()



TYPE: INSTANCE FUNCTION

PURPOSE: Read the test identifier string from a channel object.

USAGE:

IDString = Channel.Ident()

IDString STRING The channel identifier string.

## Channel.System()



TYPE: INSTANCE FUNCTION

PURPOSE: Read the name of a chemical system from a channel object. This name can be used to recover a group of parameters from the chemical system database file, SYSTEM.SET.

The Channel.System() function only recalls the system name, it does not recover the actual parameters.

USAGE:

LookupString =Channel.System()

LookupString STRING The chemical system name. Used to lookup the system in the SYSTEM.SET file.

## Channel.Area()



### *Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Read the sample area from a channel object.

**USAGE:**

Size = Channel.Area()

Size REAL The sample area.

## Channel.PrintNotes()



### *Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Prints the Channel object's notes information to the output file.

**USAGE:**

Channel.PrintNotes(Tag)

Tag STRING Optional. A string to be used as a tag for the block of text.  
If this argument is missing the Channel's tag is printed with the notes field.

## Channel.Dialog



### *Related Topics*

**TYPE:** LICENSE

**PURPOSE:** Allows a Channel to be displayed & modified in a Setup dialog box. This license also allows channel information (except Notes) to be saved/restored in a Setup file.

See the CHANNEL. Dialog license description below for a description of a Channel objects appearance in the Setup() dialog box.

## CHANNEL.Dialog

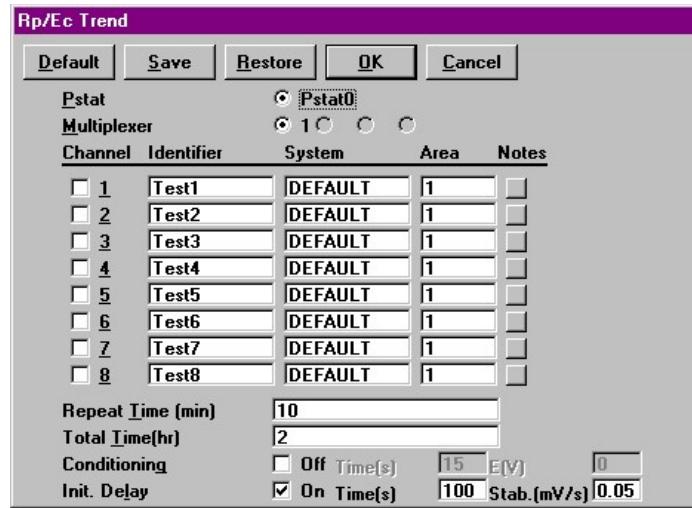


### *Related Topics*

**TYPE:** LICENSE

**PURPOSE:** This license is unusual in that it is a license for the CHANNEL class. Individual channel objects also have a dialog license. The class CHANNEL is normally passed as an argument to the Setup() function just prior to passing one or more Channel objects. A dialog box that uses this Setup() calling sequence is seen in the figure below.

**Setup() Appearance with CHANNEL and Channel Objects**



**USAGE:** The Channel section of this dialog box was produced by this code fragment.

```

result = Setup("Rp/Ec Trend",
& Pstat,
& Mux,
& CHANNEL,
& Channel[0],
& Channel[1],
& Channel[2],
& Channel[3],
& Channel[4],
& Channel[5],
& Channel[6],
& Channel[7],
& DLGSPACE.New(3),
& RepTime,
& ...

```

The CHANNEL argument to Setup() produced the line highlighted with a double underline. This line consists of labels for the columns below it. Eight Channel objects passed as arguments produce the next 8 lines. Each line has editable fields for the Active BOOL, the Ident STRING, the System STRING, the REAL area and a Notes field.

## Char()



**TYPE:** REGULAR FUNCTION

**PURPOSE:** Returns a character given a valid Ascii number.

**USAGE:**

Charact = Char(AsciiNumb)

AsciiNumb INDEX Valid Ascii number.

Charact STRING Character whose Ascii number is AsciiNumb.

## ClassIndex()



### *Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Return a unique numerical code identifying the class or type of a variable. The primary use is in checking that two items are of the same data type or class. For example,

```
if (ClassIndex(Item1) eq ClassIndex(Pstat1)
.....
```

USAGE:

```
Result = ClassIndex(Item)
```

Result INDEX A code identifying the class or data type.

Item any type The variable to be ClassIndex'ed

## ClassName()



### *Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Return a string containing the name of a variable's class or datatype.

USAGE:

```
String = ClassName(Item)
```

String STRING The name of the class for Item.

Item The object whose class name is desired

## ClassNew()



### *Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Dynamically create a class from a class name and super class. These classes are created on the fly during the execution of a script.

USAGE:

```
Result = ClassNew(Name, SuperClass)
```



Name STRING Name of the class being created

SuperClass STRING Name of the superclass this class is derived from. If missing, no superclass is used.

Result CLASS Newly created class. For this class to be accessed from other functions, the result must be assigned to a global variable.

## ClassAddCSEL()



### *Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Add a function or variable name to a class. This function is designed mainly for dynamically created classes but can be used (carefully) to modify the action of existing classes. If a selector already exists for that class, it is overwritten with the new definition. If the Value is a callin or native Explain function, the new selector is a class function, otherwise it is a class variable. After the selector has been added, it can be used like any other selector.

USAGE:

ClassAddCSEL(Class, Selector, Value)

Class CLASS The class to which the selector is being added.

Selector STRING The new selector.

Value any The default value for the selector. This is usually used to assign a method to a selector.

## ClassAddISel()



### *Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Add an instance selector, e.g. instance name, to a class. This function is designed mainly for dynamically created classes but can be used (carefully) to modify the action of existing classes. If a selector already exists for that class, it is overwritten with the new definition. If the Value is a callin or native Explain function, the new selector is an instance function, otherwise it is an instance variable. After the selector has been added, it can be used like any other selector. Objects created before the selector is added will not be modified.

USAGE:

ClassAddISel(Class, Selector, Value)

Class CLASS The class to which the selector is being added.

Selector STRING The new selector.

Value any The default value for the selector. This is usually used to assign a method to a selector.

## Complex



*Related Topics*

TYPE: CLASS

PURPOSE: The COMPLEX class is a mathematical class which allows the storage and manipulation of the real and imaginary components of complex numbers.

### COMPLEX.New()



*Related Topics*

TYPE: CLASS FUNCTION

PURPOSE: Create a new instance of COMPLEX.

USAGE:

```
Number = COMPLEX.New(Real, Imag)
```

Real REAL Real component of the complex number.

Imag REAL Imaginary component of the complex number.

Number COMPLEX A new object.

### Complex.Real()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Return the real component of a complex number.

USAGE:

```
RealComp = Complex.Real()
```

RealComp REAL Real component of complex number  
Complex.

## Complex.Imag()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Return the imaginary component of a complex number.

USAGE:

ImagComp = Complex.Imag()

ImagComp REAL Imaginary component of complex  
number Complex.

## Complex.SetReal()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Set the real component of a complex number.

USAGE:

Complex.SetReal(Number)

Number REAL New real component of Complex.

## Complex.SetImag()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Set the imaginary component of a complex number.

USAGE:

Complex.SetImag(Number)

Number REAL New imaginary component of Complex.

## Complex.Add()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Add two complex numbers. The parameter is a second complex number.

USAGE:

Sum = Complex1.Add(Complex2)

Complex1	COMPLEX	First of the 2 complex numbers to add.
Complex2	COMPLEX	Second of the 2 complex numbers to add.
Sum	COMPLEX	Sum of Complex1 and Complex2.

## Complex.Sub()



### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Subtract two complex numbers. The parameter is a second complex number. The second complex number is subtracted from the first complex number.

USAGE:

Diff = Complex1.Sub(Complex2)

Complex1 COMPLEX First of the 2 complex numbers to subtract.

Complex2 COMPLEX Second of the 2 complex numbers to subtract.

Diff COMPLEX Difference between Complex1 and Complex2.

## Complex.Mul()



### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Multiply two complex numbers. The parameter is a second complex number.

USAGE:

Product = Complex1.Mul(Complex2)

Complex1 COMPLEX First of the 2 complex numbers to multiply.

Complex2 COMPLEX Second of the 2 complex numbers to multiply.

Product COMPLEX Product of Complex1 and Complex2.

## Complex.Div()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Divide two complex numbers. The parameter is a second complex number. Complex2 is divided into Complex1.

**USAGE:**

Quotient = Complex1.Div(Complex2)

Complex1	COMPLEX	First of the 2 complex numbers to divide.
Complex2	COMPLEX	Second of the 2 complex numbers to divide.
Quotient	COMPLEX	Quotient of Complex1 and Complex2 = Complex1 / Complex2

## Complex.Con()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Return the complex conjugate of a complex number.

**USAGE:**

Conjugate = Complex.Con()

Conjugate	COMPLEX	Complex conjugate of the complex number Complex.
-----------	---------	--

## Complex.isEqual()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Compare the values two complex numbers to see if they are equal to each other. The parameter is a second complex number.

**USAGE:**

Result = Complex1.isEqual(Complex2)

Complex1	COMPLEX	First of the 2 complex numbers to compare.
----------	---------	--

Complex2 COMPLEX Second of the 2 complex numbers to compare.

Result BOOL TRUE = The Complex Numbers are Equal.  
 FALSE = The Complex Number are Not-Equal.

## Complex.Cos()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Calculate the cosine of a complex number.

USAGE:

Result = Complex.Cos()		
Complex	COMPLEX	Existing instance of a Complex Number.
Result	COMPLEX	Cosine of complex number Complex.

## Complex.Sin()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Calculate the Sine of a complex number.

USAGE:

Result = Complex.Sin()		
Complex	COMPLEX	Existing instance of a Complex Number.
Result	COMPLEX	Sine of complex number Complex.

## Complex.Tan()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Calculate the tangent of a complex number.

USAGE:

Result = Complex.Tan()		
Complex	COMPLEX	Existing instance of a Complex Number.
Result	COMPLEX	Tangent of complex number Complex.

## Complex.Mag()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Calculate the magnitude of a complex number.

USAGE:

Result = Complex.Mag()

Complex    COMPLEX

Existing instance of a Complex Number.

Result    REAL

Magnitude of complex number Complex.

## Complex.Phi()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Calculate the phase of a complex number in Radians.

USAGE:

Result = Complex.Phi()

Result = Complex.Phi()

Complex    COMPLEX

Existing instance of a Complex Number.

Result    REAL

Phase of complex number Complex in radians

## Complex.Show()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Output the value of a Complex number to the STDOUT window.

USAGE:

Complex.Show(Text)

Complex COMPLEX Existing instance of a Complex Number.

Text STRING Description to be displayed with  
Complex Number.

## Config()



*Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Lookup an INI file value.

USAGE:

Result = Config(File,Section,Key)

File	STRING	What file to look in. If the file doesn't have a path, the system looks for it in the same directory that contains the Gamry5.ini file. (Typically "C:\Documents and Settings\All Users\Application Data\Gamry Instruments\Framework\")
Section	STRING	Section name.
Key	STRING	Key name
Result	STRING	If value is found it is returned as a string
or	NIL	If value is not found, a NIL is returned.

## Cos()



*Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Calculate the cosine of an angle. The angle must be in radians

USAGE:

Result = Cos(Angle)

Angle REAL The angle of which to calculate the Cosine

Result REAL The Cosine of Angle

## Cosh()



*Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Calculate the hyperbolic cosine of a number. The definition of Cosh is:

$$\text{Cosh}(x) = [e^x + e^{-x}] / 2$$

USAGE:

Result = Cosh(Number)

Number REAL The number of which to calculate the hyperbolic cosine.

Result REAL The hyperbolic cosine of Number



## CURVE



TYPE: CLASS

The CURVE class and Curve objects do not exist as distinct Explain entities.

The terms CURVE and Curve object are used in this help section to refer to objects that acquire and hold run time data. Examples are the CPIV, CIIV, GALVCOR, and OCV classes and objects of these classes.

## DATACOL



TYPE: CLASS

Objects of the DATACOL class are used to hold a one dimensional data array. Datacol objects are usually short lived, existing only during a function call. As appropriate for transient objects, there are no Datacol functions callable directly from Explain.

Datacol objects are generally produced by a call to a Curve.DataCol() function.

## DateStamp()



TYPE: REGULAR FUNCTION

PURPOSE: Create a string of the form "Month/Day/Year". If no parameter is given, use the current system date. Otherwise, convert the parameter into a date. The parameter is derived from function Time().

USAGE:

```
Result = DateStamp()
```

or

```
Result = DateStamp(Marker)
```

Marker INDEX Seconds from 0:0:0, 1/1/70

Result STRING Date in the form "MON/DAY/YEAR"

## Dawdle()



TYPE: REGULAR FUNCTION

PURPOSE: Dawdle() freezes execution of the current script until the user keys the Skip button. One use for dawdle is a delay before

erasing the runner window so that the user has time to examine the data display.

USAGE:

Dawdle()

## DLGSPACE



*Related Topics*

TYPE: CLASS

PURPOSE: DlgSpace objects are used to generate white space in a Setup() dialog box. They are generally created right in the call to Setup().

```
Setup( Tag,
& item1,
& item2,
& DLGSPACE.New(4),
& item3,
...

```

The space is entered in "dialog box units". Each unit is significantly shorter than a line of text.

## DLGSPACE.New()



*Related Topics*

TYPE: CLASS FUNCTION

PURPOSE: Generate a new DLGSPACE object.

USAGE:

```
DlgSpace = DLGSPACE.New( Space )
    DlgSpace    DLGSPACE    The new object.
    Space       INDEX       The size of the space in dialog box units.
```

## DlgSpace.Dialog



*Related Topics*

TYPE: LICENSE

PURPOSE: Informs Setup() what to do with the DlgSpace object. A DlgSpace object appears in Setup() as a vertical gap between other object displays.

## DtoR()



*Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Convert an angle from Degrees to Radians.

USAGE:

Radians = DtoR(Degrees)

Degrees REAL The angle in Degrees.

Radians REAL The angle in Radians.

**Error()***Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Indicate a fatal error to the operator. The error is serious enough that the script must be terminated. End the experiment.

USAGE:

Error(Message)

Message STRING Text to be displayed

**Execute()***Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Launch another Explain Script from within Explain. As soon as the other script is launched, the parent script continues its execution. Any number of Explain primitive types such as integers, strings and reals can be passed using this function. Objects, however, cannot be passed from one script to another.

USAGE:

Execute("Filename",Param1, Param2, . . . , ParamN)

Filename TEXT A valid path to an EXP file

ParamN UNKNOWN Any valid Explain primitive which you want to be passed to the launching script. Any number of parameters may be passed.

**ExecWait()***Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Launch another Explain Script from within Explain. As soon as the other script is launched, the parent script suspends execution until the launched script is complete. Any number of Explain primitive types such as integers, strings and reals can be passed using this function. Objects, however, cannot be passed from one script to another.

USAGE:

ExecWait("Filename",Param1, Param2, . . . , ParamN)

Filename TEXT A valid path to an EXP file

ParamN UNKNOWN Any valid Explain primitive which you want to be passed to the launching script. Any number of parameters may be passed.

**Exp()***Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Calculate  $e^x$ .

USAGE:

Result = Exp(Number)

Number REAL The exponent of e

Result REAL The result of  $e^{\text{Number}}$

**FindClassName()***Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Return a CLASS given the class name as a string. This function is used to return a class given information about its name. A specific example where this function is used is in creating a PSTAT object. As discussed in PSTAT.New, there actually are two different potentiostat classes, PC3 and PC4. The class information for each potentiostat installed in a system is stored in the GAMRY.INI file. By first reading the Class from the INI file into a STRING and then by calling the FindClassName() function, a PC3 or PC4 class can be returned.

USAGE:

Class = FindClassName(ClassName)

Class CLASS The class with the name ClassName

ClassName STRING The name of the class which you want to be returned

**Headline()**

**Related Topics**

TYPE: REGULAR FUNCTION

PURPOSE: Write a string in the Headline region of the runner screen above the real time curve display. The string is normally used to tell the operator which experiment is currently running.

USAGE:

Headline(Text)

Text STRING

## ICONST

**Related Topics**

TYPE: CLASS

PURPOSE: The ICONST class describes a constant current waveform which can be applied by a potentiostat while in the Galvanostat mode. The Iconst object encapsulates information about the applied current, the elapsed time, and the data acquisition rate. Most of the activity associated with the Iconst happens in the background while a Ciiv.Run() is being executed.

The current range used for the Iconst signal generator is selected from the absolute current requested. The current range is fixed throughout the experiment.

## ICONST.New()

**Related Topics**

TYPE: CLASS FUNCTION

PURPOSE: Create a new constant current signal generator.

USAGE:

```
Iconst = ICONST.New(Tag, Pstat, Current, Time, SampleTime)
  Tag          STRING      Object tag
  Pstat        PSTATCLASS  Potentiostat which will use the signal
  Current      REAL         Applied current in amps
  Time         REAL         Total Time in seconds
  SampleTime  REAL         Time between data acquisition steps
  Iconst       ICONST      The object created
```

## Iconst.Tweak()

**Related Topics**

TYPE: INSTANCE FUNCTION

PURPOSE: Change the parameters of an existing constant current signal generator.

Iconst.Tweak(Pstat, Current, Time, SampleTime)

Pstat	PSTATCLASS	Potentiostat which will use the signal
Current	REAL	Applied current in amps
Time	REAL	Total Time in seconds
SampleTime	REAL	Time between data acquisition steps
Iconst	ICONST	The existing Iconst object

## Index()



### Related Topics

TYPE: REGULAR FUNCTION

PURPOSE: Produce an INDEX from some other datatype.

If the quantity converted is a REAL, the number is truncated at the decimal point and converted to an INDEX (signed 32 bit integer). REAL values greater than  $(2^{31}-1)$  (approximately 2.15E9) are converted to  $2^{31}-1$ . Correspondingly, REAL values more negative than  $-2^{31}$  are converted to  $-2^{31}$ .

Some other data types can be converted as well. BITS are converted using 2's compliment arithmetic. If the most significant bit of a BITS quantity is a one, the resulting INDEX will be negative. BOOL quantities convert as a one for TRUE and a zero for FALSE. A NIL converts to a zero.

The INDEX() function acting on a STRING is bounded similarly to REALS. If the string does not contain a number, the return value is indeterminate.

The INDEX() function applied to a complex data type such as an objects will return a meaningless number.

USAGE:

Converted = Index(Item)

Converted INDEX The result of the conversion.

Item REAL or The variable to be converted.

BITS  
 BOOL  
 NIL  
 STRING

## IQUANT



### Related Topics

TYPE: CLASS

PURPOSE: The IQUANT class allows a user to edit, save and restore INDEX (integer) values in Setup(). Remember, INDEX's by themselves, cannot be given as parameters to Setup() since Explain uses Call by Value to pass parameters. An INDEX must therefore be encapsulated in a IQUANT to be used in Setup.

## IQUANT.New()

**Related Topics**

TYPE: CLASS FUNCTION

PURPOSE: Create and initialize a new Iquant object.

USAGE:

```
Iquant = IQUANT.New(Tag, Quantity, Prompt)
```

Tag STRING Used by Setup() Save/Restore,  
Tag.Printl()

Iquantity INDEX Initial value for Iquant

Prompt STRING Used by Setup() to prompt operator for  
input

Iquant IQUANT The initialized object

## Iquant.Printl()

**Related Topics**

TYPE: INSTANCE FUNCTION

PURPOSE: Print a Iquant to the current Output file. The Iquant will be printed in the format:

```
Tag <tab> IQUANT <tab> Quantity <tab> Prompt
```

If the Tag is empty, Iquant.Printl() will print only the Quantity.

For example, if the Iquant was created with:

```
Iquant = IQUANT.New("LOOPS", 120, "Loops")
```

then Iquant.Printl() would give

```
LOOPS <tab> IQUANT <tab> 120<tab>Loops
```

USAGE:

```
Iquant.Printl(Tag)
```

Tag STRING Optional. Used to print a different Tag

than the one used in the call to `.New`

## Iquant.Value()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Copy the current value of a Iquant into an INDEX.

USAGE:

```
Value = Iquant.Value()
```

Value INDEX Current value of the Iquant

## Iquant.SetValue()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Set the value of the Iquant.

USAGE:

```
Iquant.SetValue(Value)
```

Value INDEX New value for the Iquant.

## Iquant.Sprint()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Print the Iquant to a string.

USAGE:

```
Result = Iquant.Sprint(Format)
```

Format STRING (Optional) C-language format specifier  
for long integer.  
Default is %ld.

Result STRING Resulting formatted number.

## IQUANT.Dialog



*Related Topics*



[TYPE:](#) LICENSE

[PURPOSE:](#) Allow an Iquant to be edited, saved, and restored in Setup(). The Iquant will be edited using the fields:

Prompt [Quantity ]

When the operator enters a value, Setup will attempt to turn it into a valid INDEX and replace the Value in Iquant with it. If Setup() cannot translate a string into an INDEX, it will default to the value 0 (not NIL).

## IsUpper()



[TYPE:](#) REGULAR FUNCTION

[PURPOSE:](#) Determine whether an Ascii number refers to an Uppercase character (A - Z).

[USAGE:](#)

Result = IsUpper(AsciiNumb)

AsciiNumb INDEX Ascii number of character in question.

Result BOOL TRUE = Character is Uppercase Letter.

FALSE = Character is not Uppercase  
Letter.

## IsLower()



[TYPE:](#) REGULAR FUNCTION

[PURPOSE:](#) Determine whether an Ascii number refers to a Lowercase character (a - z).

[USAGE:](#)

Result = IsLower(AsciiNumb)

AsciiNumb INDEX Ascii number of character in question.

Result BOOL TRUE = Character is Lowercase Letter.

FALSE = Character is not Lowercase  
Letter.

## IsAlpha()



[TYPE:](#) REGULAR FUNCTION

[PURPOSE:](#) Determine whether an Ascii number refers to a Letter (A - Z) or (a - z).

[USAGE:](#)

Result = IsAlpha(AsciiNumb)

AsciiNumb INDEX Ascii number of character in question.

Result BOOL TRUE = Character is a Letter.  
FALSE = Character is not a Letter.

## IsAlnum()



*Related Topics*

[TYPE:](#) REGULAR FUNCTION

[PURPOSE:](#) Determine whether an Ascii number refers to an Alphanumeric character (A - Z), (a - z), or (0 - 9).

[USAGE:](#)

Result = IsAlnum(AsciiNumb)

AsciiNumb INDEX Ascii number of character in question.

Result BOOL TRUE = Character is Aphanumeric.  
FALSE = Character is not Alphanumeric.

## IsDigit()



*Related Topics*

[TYPE:](#) REGULAR FUNCTION

[PURPOSE:](#) Determine whether an Ascii number refers to an numeric character (0 - 9).

[USAGE:](#)

Result = IsDigit(AsciiNumb)

AsciiNumb INDEX Ascii number of character in question.

Result BOOL TRUE = Character is numeric.  
FALSE = Character is not numeric.

## IsXdigit()



*Related Topics*

[TYPE:](#) REGULAR FUNCTION

[PURPOSE:](#) Determine whether an Ascii number refers to a hexadecimal digit (A - F), (a - f), or (0 - 9).

[USAGE:](#)

Result = IsXdigit(AsciiNumb)

AsciiNumb INDEX Ascii number of character in question.

Result BOOL TRUE = Character is hexadecimal digit.  
FALSE = Character is not hexadecimal digit.

## IsSpace()



TYPE: REGULAR FUNCTION

PURPOSE: Determine whether an Ascii number refers to white-space. Returns TRUE if AsciiNumb is a white-space character (0x09 - 0x0D or 0x20). Each of these routines returns FALSE if AsciiNumb does not satisfy the test condition.

USAGE:

Result = IsSpace(AsciiNumb)

AsciiNumb INDEX Ascii number of character in question.

Result BOOL TRUE = Character is white-space.  
FALSE = Character is not white-space.

## IsPunct()



TYPE: REGULAR FUNCTION

PURPOSE: Determine whether an Ascii number refers to a punctuation character. Returns TRUE for any printable character that is not a space character or an alphanumeric character.

USAGE:

Result = IsPunct(AsciiNumb)

AsciiNumb INDEX Ascii number of character in question.

Result BOOL TRUE = Character is punctuation.  
FALSE = Character is not punctuation.

## IVT



TYPE: CLASS

**PURPOSE:** The IVT class manages data acquisition for a variety of curves at a reduced level of control. This general purpose curve is used for conditioning as well as in several add-on packages. The IVT class contains controlled potential and controlled current I-V curves and a version of galvanic corrosion. There may be more than one data acquisition object in a script but only one may be running (i.e. acquiring values) at any time. The typical life cycle of an IVT object is:

```
Pstat.SetCtrlMode(PstatMode) ; Set Potentiostatic Mode
Ivt = IVT.New(Tag, Pstat) ; Create the IVT object
Ivt.SetPlotView(. . .) ; Set the plot type
Ivt.Run() ; Run the curve
Ivt.PrintI() ; Print the curve out to Output
```

## IVT.New()



### Related Topics

**TYPE:** CLASS FUNCTION

**PURPOSE:** Create a new Ivt object.

**USAGE:**

```
Ivt = IVT.New(Tag, Pstat)
```

Tag STRING Curve tag for data output. Used to label the file

Pstat Object Pstat is an object with a PSTAT license

Ivt IVT The curve object

## Ivt.Run()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Run the curve. When Ivt.Run() is executed, the acquired data is displayed in a real-time curve. There are four buttons active during Ivt.Run() - Abort, Pause/Continue, Skip, and Point. The Abort button terminates the entire experiment. The Skip button skips to the next phase of the experiment, either to the end of the curve or to the next region of the signal generator's operation.

The active run can be halted in several ways 1) when the signal generator is finished, 2) if the operator hits a Skip Button, or 3) if one of the stopping criteria is met. The stopping criteria can be enabled or disabled using the Ivt.StopAt functions.

If the Pstat.SetIrruptMode() switch is TRUE, the system will use a current interrupt process to measure the uncompensated IR drop throughout the experiment.

**USAGE:**

```
Result = Ivt.Run()
```

Result	INDEX	
NO_STOP		Normal termination or Skip/F2 button pressed
STOP_V_UNDER		E<Emin
STOP_V_OVER		E>Emax
STOP_I_UNDER		I<Imin
STOP_I_OVER		I>Imax

```
STOP_A_UNDER  A<Amin
STOP_A_OVER   A>Amax
```

## Ivt.Printl()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Print the results of a Ivt experiment out to the Output file. The Output record will be printed in the following format:

```
Tag <tab> "TABLE" <tab> Points
<tab> "Pt" <tab> "T" <tab> "Vf" <tab> "Im" <tab> "Vu" <tab> "Sig" <tab>"Ach" <tab>
  "Over"
<tab> "#" <tab> "s" <tab> "v" <tab> "A" <tab> "v" <tab> "v" <tab>"v" <tab> "bits"
<tab> Pt1 T V I V V V Flags
<tab> Pt2 T V I V V V Flags
...
<tab> PtN T V I V V V Flags
```

Where the symbols have the following meanings:

Tag	The curve Tag (from <a href="#">Ivt.New()</a> or optional argument)
Points	Total number of points acquired
Pt(#)	Sequential number of a point.
T	Elapsed time from starting from <a href="#">Ivt.Run</a>
Vf	Applied Voltage vs Eref.
Im	Measured cell current.
Vu	Uncompensated Voltage
Sig	Signal sent to Control Amp
Ach	Aux Channel Voltage
Over	Overload flags

**USAGE:**

```
Ivt.Printl(Tag)
Tag           STRING           Optional. Used to print a different
                          Tag than the one used in the call to
                          Ivt.New()
```

## Ivt.StopAt()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Enable or disable the Ivt termination limits. There is an upper and lower limiting condition for each channel. A curve may prematurely terminate upon exceeding one of these limits.

**USAGE:**

Ivt.StopAt(Imin, Imax, Vmin, Vmax, Amin, Amax)

Imin	REAL	Lower limit on I channel
	or NIL	Disable I channel lower limit
Imax	REAL	Upper limit on I channel
	or NIL	Disable the I channel upper limit
Vmin	REAL	Lower limit on V channel
	or NIL	Disable V channel lower limit
Vmax	REAL	Upper limit on V channel
	or NIL	Disable V channel upper limit
Amin	REAL	Lower limit on Aux channel
	or NIL	Disable Aux channel lower limit
Amax	REAL	Upper limit on Aux channel
	NIL	Disable Aux channel upper limit

## Ivt.StopAtDelay()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Used to avoid premature stopping of a curve caused by noise creating a false reading for a StopAt test. Avoids false readings by requiring that the StopAt criterion be met for a number of data points (N) before the curve stops. The parameter N can be individually set for each StopAt test.

For absolute tests, such as  $V > \text{Limit}$ , the criterion must be met for N points consecutively before the criterion will be accepted. One point missing the criterion limit resets the N point counter.

USAGE:

Ivt.StopAtDelay(Imin, Imax, Vmin, Vmax, Amin, Amax)

Imin INDEX Delay for this number of points  
or  
NIL Only one point required

Imax INDEX Delay for this number of points  
or  
NIL Only one point required

Vmin INDEX Delay for this number of points  
or  
NIL Only one point required

Vmax INDEX Delay for this number of points  
or  
NIL Only one point required

Amin INDEX Delay for this number of points  
or  
NIL Only one point required

Amax INDEX Delay for this number of points  
or  
NIL Only one point required

## Ivt.Thresh()



*Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Setup threshold conditions which must be met before Ivt.StopAt() tests will be performed. Once any of the threshold tests is met, the StopAt criteria are evaluated for the rest of the scan. There are six threshold tests that you can perform:

- |    |      |                    |
|----|------|--------------------|
| 1. | IMin | $I < I\_MinThresh$ |
| 2. | IMax | $I > I\_MaxThresh$ |
| 3. | EMin | $E < E\_MinThresh$ |
| 4. | EMax | $E > E\_MaxThresh$ |
| 5. | Amin | $A < A\_MinThresh$ |
| 6. | Amax | $A > A\_MinThresh$ |

Threshold tests allow you to tune the StopAt tests more carefully. For example, you can avoid a premature experiment termination due to an initial transient.

In general you will only enable one of the threshold tests. The others will be turned off by a NIL argument in the function call.

Note that any threshold test starts all StopAt tests. Also once StopAt testing is enabled, it is never disabled, even if the Threshold test that enabled it is not longer valid.

**USAGE:**

Ivt.Thresh(IMin, IMax, EMin, EMax, Amin, AMax)		
IMin	REAL	Enable StopAt if $I < IMin$
	NIL	Disable this test
IMax	REAL	Enable StopAt if $I > IMax$
	NIL	Disable this test
EMin	REAL	Enable StopAt if $E < EMin$
	NIL	Disable this test
EMax	REAL	Enable StopAt if $E > EMax$
	NIL	Disable this test
Amin	REAL	Enable StopAt if $A < Amin$
	NIL	Disable this test
AMax	REAL	Enable StopAt if $A > AMax$
	NIL	Disable this test

## Ivt.DataCol()



*Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Create and return an object of the class DATACOL. The object is filled with data extracted from one column of the IVT object's data array. The data, once extracted, can then be submitted to other objects for further calculation. For example, a LINFIT object performs a linear regression analysis on two DATACOL objects.

A new DATACOL object is automatically created by the call to IVT.DataCol(). It is usually transient, existing only for one function call.

**USAGE:**

DataCol = Ivt.DataCol(ColumnCode)

DataCol DATACOL The new object containing the data extracted from the IVT data array.

ColumnCode INDEX A code identifying the column to be used. The following Predefined constants should be used:

*IVT\_T* = Time  
*IVT\_Vf* = Applied Voltage  
*IVT\_Vu* = Uncompensated Voltage  
*IVT\_Im* = Measured Current  
*IVT\_Am* = Measured Voltage Aux Channel  
*IVT\_Sig* = Signal sent to Control Amp

## Ivt.DataValue()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Extract a single data item from a IVT object. The data item is taken from the ith data point and the jth data column. If no data has been taken the returned value is NIL.

**USAGE:**

Data Value = Ivt.DataValue(PointNum,ColNumber)

Result REAL The returned value.  
 or  
 NIL No data acquired yet.

PointNum INDEX Point number, zero based. If PointNum is out of range, a value from the last or acquired point is returned.

NIL Always return a value from the last acquired point.

ColNumber INDEX A code identifying the column to be used. The following Predefined constants should be used:

*IVT\_T* = Time  
*IVT\_Vf* = Applied Voltage  
*IVT\_Vu* = Uncompensated Voltage  
*IVT\_Im* = Measured Current  
*IVT\_Am* = Measured Voltage Aux Channel  
*IVT\_Sig* = Signal sent to Control Amp

## Ivt.Count()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Return the number of points actually acquired in a IVT object.

**USAGE:**

NumPoints = Ivt.Count()

NumPoints INDEX The number of points in the IVT object.  
 Zero is returned if no data has been acquired yet.



## Ivt.SetPlotView()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Format the real time display of an IVT curve. The X-Axis source is fixed to time. X-Axis units are always seconds (s). The Z-Axis parameters are optional. If included a double plot will be shown, otherwise a single plot will be displayed.

**USAGE:**

Ivt.SetPlotView( XMinVal, XLabel, YSource, YMinVal, YLabel, YUnits, ZSource, ZMinVal, ZLabel, ZUnits )

XMinVal REAL Minimum resolution for X-Axis

XLabel STRING X-Axis label

YSource INDEX Column number to be plotted on Y-Axis

A code identifying the column to be used. The following predefined constants should be used:

<i>IVT_T</i>	= Time
<i>IVT_Vf</i>	= Applied Voltage
<i>IVT_Vu</i>	= Uncompensated Voltage
<i>IVT_Im</i>	= Measured Current
<i>IVT_Am</i>	= Measured Voltage Aux Channel
<i>IVT_Sig</i>	= Signal sent to Control Amp

YMinVal REAL Minimum resolution for Y-Axis

YLabel STRING Y-Axis label

YUnits STRING Y-Axis units

ZSource INDEX Column number to be plotted on Z-Axis  
Same as YSource (optional)

ZMinVal REAL Minimum resolution for Z-Axis (optional)

ZLabel STRING Z-Axis label (optional)

ZUnits STRING Z-Axis units (optional)

## Ivt.Activate()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Make the real time plot of the Ivt object the Active plot. If the Runner window Curve List control is set to Active, the plot will be displayed. If the Runner window Curve List control is set to a definite curve, the display does not change.

**USAGE:**

Ivt.Activate()

## Ivt.Sprint()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Print the status of the Ivt to a string. The object's tag and last acquired data point will be printed in the format:

Tag,PointNumber

**USAGE:**

Result = Ivt.Sprint()

Result STRING Resulting tag & point number (0 based).

## Ivt.SprintPoint()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Used in output of data from a IVT object. A data point from the object is printed to an ASCII string. This string can in turn be printed to the active output file.

The format of the data string is:

PointNumber <tab> T <tab> Vf <tab> Im <tab> Vu <tab> Sig <tab> Ach <tab> Over

You can use this function along with the Printl() function to output point i to the output file as follows:

```
Printl("\t",Ivt.SprintPoint(i))
```

**USAGE:**

Result = Ivt.SprintPoint(PointNumber)

Result                    STRING

PointNumber            INDEX

The output string

The point to be printed. PointNumber is 0 based. If the requested point has not been taken, a parameter error will be issued.

T                        REAL

Elapsed time from starting from

[Ivt.Run](#)

Vf                       REAL

Applied E vs Eref.

Im                       REAL

Measured cell current.

Vu                       REAL

Uncompensated Voltage

Sig                      REAL

Signal sent to Control Amp

Ach                      REAL

Aux Channel Voltage

Over                     STRING

Hexadecimal number representing overload status

## LABEL



### Related Topics

**TYPE:** CLASS

**PURPOSE:** Class LABEL defines a user editable string label that may be used for a filename, a warning string, a status string, an

experiment label, etc. To create a string editable in Setup, use the class LABEL instead of the simple datatype, STRING.

## LABEL.New()



### *Related Topics*

TYPE: CLASS FUNCTION

PURPOSE: Create a new label. Suppose a LABEL object is created with the statement:

```
ExptID = LABEL.New("EXPERIMENT", 20, "17S304", "Edit Expt. ID")
```

ExptID is an object of class LABEL with a value of "17S304". "17S304" is the portion of the ExptID object that can be modified by the user. It can be as many as 20 characters long.

LABEL.New() will fail and stop the script if an error occurs.

USAGE:

```
Label = LABEL.New(Tag, Length, Value, Prompt)
```

Tag STRING Tag used to identify this object when writing, saving, or restoring

Length INDEX Size of field in bytes used to store value

Value STRING The string field that can be edited, written, saved or restored

Prompt STRING Prompt used in Setup dialog box when this object is being edited

Label LABEL Label object created

## Label.Print()



### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Print the Label to the current Output. The object created will be printed in the form:

```
Tag <tab>LABEL <tab> Value <tab> Prompt
```

being appended to the output file. Most data bases can resolve this into two fields by using the tab character as a field separator.

USAGE:

```
Label.Print(Tag)
```

Tag STRING Optional. Used to print a different Tag than the one used in the call to "Object."

## Label.SetValue()



### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Allows the Script to set a label's string from within Explain.

USAGE:

```
Label.SetValue(Test)
```

Text STRING The new text for the Label's data string.

## Label.Value()



### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Copy the Label text into a STRING. For example, with the sample Label object described above:

```
String = ExptID.Value()
```

would create a STRING with the contents "17S304".

USAGE:

```
Text = Label.Value()
```

Text STRING contents = Value field of Label

## Label.Sprint()



### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Print a label object out to a string.

USAGE:

```
Result = Label.Sprint(Format)
```

Format STRING (Optional) C-Language format string.  
Uses %.127s as default.

## LABEL.Dialog



### *Related Topics*

TYPE: LICENSE

**PURPOSE:** To allow LABEL objects to be edited, saved, and restored in Setup(). Setup() will format a LABEL object as:

Prompt: [Value ]

where the Value field is editable. It will save and restore the object indexing it with the Tag information.

## LineOpt()



### *Related Topics*

**TYPE:** REGULAR FUNCTION

**PURPOSE:** Return a sample time that will result in successive samples 180 degrees out of phase with the AC power line frequency. Averaging an even number of samples recorded 180 degrees out of phase with the line rejects noise at the AC line frequency.

The standard scripts use this function to control sample timing in the initial delay portion of the experiment. This improves the accuracy of Eoc measurements with high impedance reference electrodes.

This function only works properly if LineFreq field in the "GAMRY.INI" file is set for the proper AC line frequency.

**USAGE:**

NewTime = LineOpt(OldTime)

NewTime REAL The time calculated to result in successive sample that are out of phase with the AC power line. This time is as close as possible to OldTime.

OldTime REAL The time used as the basis for the calculation. The calculated time is kept as close as possible to the Old Time.

## LoadLibrary()



### *Related Topics*

**TYPE:** REGULAR FUNCTION

**PURPOSE:** Dynamically load a DLL at run time. This is the equivalent to the statement:

library Result = LibraryName

**USAGE:**

Result = LoadLibrary(LibraryName)

LibraryName STRING Library file name. System searches

- 1) current directory
- 2) Windows directory
- 3) Windows system directory
- 4) Gamry Framework ProgDir
- 5) PATH directories

Result LIBRARY This is the library variable. It must be declared global to be accessible in other

functions.

**Log()**



TYPE: REGULAR FUNCTION

PURPOSE: Calculate the natural (base e) logarithm of a number. If Number < 0, the absolute value is taken first.

USAGE:

Result = Log(Number)

Number REAL Number to find the log of.

Result REAL Result of the calculation.

**Log10()**



TYPE: REGULAR FUNCTION

PURPOSE: Calculate the base 10 logarithm of a number. If Number < 0, the absolute value is taken first.

USAGE:

Result = Log10(Number)

Number REAL Number to find the log of.

Result REAL Result of the calculation

**Mean()**



TYPE: Regular Function

PURPOSE: Calculate statistical mean on a set of data. The data set can include all of the data in a DATACOL object, or a subset of the data in a DATACOL object.

USAGE:

Result = Mean (DataCol, FirstPoint, LastPoint)

DataCol	DATACOL	The column of data upon which to calculate the statistics
FirstPoint	INDEX	The first point to include in the data set (Optional)
LastPoint	INDEX	The last point to include in the data set (Optional)
Result	REAL	The Mean of the DataCol

Sum(X)/N

## MessageBox()



### Related Topics

**TYPE:** Regular Function

**PURPOSE:** The MessageBox function creates, displays, and operates a message box. The message box contains an application-defined message and title, plus any combination of predefined icons and push buttons.

**USAGE:**

Result = MessageBox (Title, Message, Style)

Title STRING Text to be displayed in Title bar of the message box.

Message STRING Text to be displayed in the body of the message box.

Style BITS Combination of styles which will be used to define the look and function of the message box.

0x00000000 = MB\_OK  
 0x00000001 = MB\_OKCANCEL  
 0x00000003 = MB\_YESNOCANCEL  
 0x00000004 = MB\_YESNO  
 0x00000010 = MB\_ICONHAND  
 0x00000020 = MB\_ICONQUESTION  
 0x00000030 = MB\_ICONEXCLAMATION  
 0x00000040 = MB\_ICONASTERISK  
 0x00000030 = MB\_ICONWARNING  
 0x00000010 = MB\_ICONERROR  
 0x00000040 = MB\_ICONINFORMATION  
 0x00000010 = MB\_ICONSTOP  
 0x00000000 = MB\_DEFBUTTON1  
 0x00000100 = MB\_DEFBUTTON2  
 0x00000200 = MB\_DEFBUTTON3  
 0x00000300 = MB\_DEFBUTTON4

Result REAL Result of the calculation.  
 1 = IDOK  
 2 = IDCANCEL  
 3 = IDABORT  
 4 = IDRETRY  
 5 = IDIGNORE  
 6 = IDYES  
 7 = IDNO

## Modulus()



### Related Topics

**TYPE:** Regular Function

**PURPOSE:** Calculate modulus of a complex number. Modulus =  $\sqrt{\text{Re}*\text{Re} + \text{Im}*\text{Im}}$ .

**USAGE:**

Result = Modulus(RealPart, ImagPart)

RealPart REAL Real part of the complex number.

ImagPart REAL Imaginary part of the complex number.

Result REAL Result of the calculation.

## MUX



*Related Topics*

TYPE: CLASS

PURPOSE: The MUX class encapsulates the functions and data required for control of an ECM8 Electrochemical Multiplexer by an Explain script. The functions within this class have a similar functionality to those in the C Language Library provided with the ECM8. The underlying code is somewhat different however.

An object of the MUX class is not useful without a corresponding ECM8 connected to the computer. This ECM8 must be configured using the Framework setup.

### MUX.New()



*Related Topics*

TYPE: CLASS FUNCTION

PURPOSE: Create a new MUX object. This object is a software construct that must be associated with a physical ECM8 before it is useful.

Similar to Pstat objects, a Mux object does not need to be associated with specific ECM8 hardware when MUX.NEW() is called. If a system contains more than one ECM8, the user can select among the available multiplexers in the Setup() function's dialog box.

The associated ECM8 is identified by a MuxNo variable.

USAGE:

MUX.New(Tag, MuxNo, Prompt)

Tag STRING Used in Save/Restore, File Output to identify and label the Mux object.

MuxNo INDEX Indicates which physical ECM8 is associated with the Mux object. Range is 1 to 4.

or

NIL Indicates no ECM8 associated yet. The ECM8 will be selected in Setup().

Prompt STRING Used in Setup() to identify the ECM8 selection line.

### Mux.Open()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Acquire the use of an ECM8. The ECM8 associated with the Mux object must be connected and operational when this function is called.

This function must be called prior to calling any other Mux object ifunctions.

Mux.Open() also initializes the ECM8 hardware to the following settings:



- All channels are turned off.
- The off mode for all channels is set to open.
- All channel DACs are set to 0.000 volts.

Mux.Open(false) will initialize the ECM8 hardware without changing the above settings.

Mux.Open() can fail for a variety of reasons. These include:

- Invalid MuxNumber setting.
- ECM8 not configured in GAMRY.INI file.
- ECM8 not operational (power must be on).
- RS232 communications failed (wrong baud rate, no cable etc.).

#### USAGE:

```
Status = MUX.Open()
```

Status BOOL Success flag.  
TRUE if Mux opened OK.  
FALSE if function failed.

### **Mux.Close()**



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Release communications with multiplexer. Closing the Mux frees up the COM port and its hardware interrupt. A closed Mux object may be reopened using Mux.Open().

This function does not reinitialize the ECM8. All ECM8 settings are left in their current state.

#### USAGE:

```
Mux.Close()
```

### **Mux.Version()**



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Report the version number of the ECM8 firmware.

#### USAGE:

```
RevNumber = Mux.Version()
```

RevNumber INDEX Version Number.

### **Mux.SetCell()**



**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Change the active cell setting. Usually used to connect one cell to the system potentiostat so that a measurement can be made on that cell. This function can also be used to make all cells inactive.

Any previously active cell is switched to its inactive cell control mode.

The Aux A/D setting is not changed by this function.

**USAGE:**

Previous = Mux.SetCell(Channel)

Channel INDEX Number from 0 to 7. One less than the new Active Cell.

For example, Channel =3 activates ECM8 cell channel 4.

or

NIL All cells are inactive.

Previous INDEX Channel number for previous setting (0 to 7).

or

NIL No channel was previously active

## Mux.Cell()



**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Report the number of the Active Cell (if any).

**USAGE:**

Active = Mux.Cell()

Active INDEX Channel number of Active Cell (0 to 7).

or

NIL All cells are inactive.

## Mux.SetAux()



**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Change the Aux A/D setting. Usually used to connect Aux A/D channel to the PC3 Potentiostat Aux A/D input BNC connector so that a measurement can be made on that channel. This function can also be used to make all Aux A/D channels inactive.

The Active Cell setting is not changed by this function.

**USAGE:**

Previous = Mux.SetAux(Channel)

Channel INDEX Number from 0 to 7. One less than the new Aux A/D channel.

or

NIL All Aux A/D channels off.

Previous INDEX Channel number for previous setting (0 to 7)

or

NIL No channel was previously on.

## Mux.Aux()



### *Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Report the Aux A/D setting.

**USAGE:**

Channel = Mux.Aux()

Channel INDEX Channel number of connected Aux A/D (0 to 7)

or

NIL All Aux A/D channels are off.

## Mux.SetOffMode()



### *Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Set the Inactive Cell control mode for one of more ECM8 multiplexer channels. The Inactive Cell control mode will be applied to a channel whenever that channel is not the Active Cell.

The control mode of currently inactive cells changes immediately. If the control mode is set on the current Active Cell, it does not become effective until the cell becomes inactive.

One of 3 possible settings are allowed. Each setting has a predefined constant that can be used to make the function call more readable. These constants are:

MuxOffOpen {0} Inactive Cell is at open circuit.

MuxOffLocal {1} Inactive Cell is controlled by local potentiostat.

MuxOffShort {2} Working and Counter electrodes of Inactive Cell are shorted. Used for Galvanic Corrosion.

**USAGE:**

Mux.SetOffMode(Channel, Mode)

Channel INDEX Apply Mode to given channel number (0-7)

or

NIL Apply Mode to all channels

or

BITS Apply Mode to channels having a one in bit pattern.  
 Bit N corresponds to channel N+1.  
 For example: 0x05 applies Mode to channels 1 and 3.  
 with all other channels unchanged.

Mode INDEX Value from 0 to 2. Predefined constants are:

MuxOffOpen {0} Open circuit.  
 MuxOffLocal {1} Local p'stat.  
 MuxOffShort {2} W.E. & C.E. are shorted.

## Mux.OffMode()



*Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Report Inactive Cell control mode of one selected ECM8 channel.

**USAGE:**

Setting = Mux.OffMode(Channel)

Channel INDEX Channel number (0-7).

Setting INDEX Offmode (0, 1, or 2) See MuxSetOffMode() for the meaning of the codes.

## Mux.SetDac()



*Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Set the local DAC (digital to analog converter) voltage for a given channel or channels. This voltage is applied to local potentiostat and appears (sign inverted) on the ECM8's rear panel output BNC(s).

The local potentiostat will only be connected to the cell if the Inactive Cell control mode has been set to the local potentiostat setting. The Mux.SetDac() function does not affect this Inactive Cell control setting.

**USAGE:**

Mux.SetDac(Channel, Voltage)

Channel INDEX Change DAC on given channel number (0-7).

or

NIL Change DAC on all channels.

or

BITS Change DAC on all channels having a one in bit pattern.  
 Bit N corresponds to channel N+1.  
 For example: 0x05 changes DAC on channels 1 and 3  
 with all other channels unchanged.

Voltage REAL Must be a number between  $\pm 5.12$  V.

## Mux.Dac()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Report the voltage on a given channel's DAC

USAGE:

Voltage = Mux.Dac(Channel)

Channel INDEX Channel number (0-7)

Voltage REAL Voltage. Range  $\pm 5.12$  V.

## Mux.Dialog



*Related Topics*

TYPE: LICENSE

PURPOSE: Allow selection of an ECM8 multiplexer in a Setup() dialog box. The selected ECM8 will be associated with the Mux object following a call to Mux.Open().

The Mux object is displayed as a set of "radio buttons":

Prompt  1  2  3  4

Only one radio button can be selected at a time. The user may modify the setting by selecting any of the labeled buttons. If a Mux number is unavailable, the corresponding button will be visible but will not be labeled.

## NOTES



*Related Topics*

TYPE: CLASS

PURPOSE: Class NOTES defines a string, editable in Setup, that may be used for experiment notes. Class NOTES differs from Class LABEL in the length of the string allowed, the editing mode in Setup(), and the way it is printed. Notes objects can have multiple lines separated by carriage returns.

## NormalD()



*Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Normalize an angle in Degrees from -360 to 360.

USAGE:

Normalized = NormalD(Degrees)

Degrees REAL The original angle in Degrees.

Normalized REAL The angle normalized from -360 to 360 degrees.

## NormalR()



### *Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Normalize an angle in Degrees from  $-2\pi$  to  $2\pi$ .

USAGE:

Normalized = NormalR(Radians)

Radians REAL The original angle in Radians.

Normalized REAL The angle normalized from  $-2\pi$  to  $2\pi$ .

## NOTES.New()



### *Related Topics*

TYPE: CLASS FUNCTION

PURPOSE: Create a new object of class NOTES. Suppose the object is created with the statement:

```
Notes = NOTES.New("CONDITIONS", 400, NIL, "Notes...")
```

Notes is an object of class NOTES. Initially its value is NIL which NOTES.New turns into an empty string. The field can hold up to 400 characters.

We'll use the example above throughout the class description.

USAGE:

```
Notes = NOTES.New(Tag, Length, Value, Prompt)
```

Tag STRING Tag used to identify this object when writing, saving, or restoring

Length INDEX Size of field in bytes used to store value.

Value STRING The string field that can be edited, written, saved or restored

Prompt STRING Prompt used in Setup dialog box when this object is being edited.

Notes NOTES Notes object created

NOTES.New() will fail and stop script if an error occurs. Errors might include invalid arguments or insufficient memory.

## Notes.Printl()



### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Print Notes to the output file. The object is printed in the format:

```
Tag <tab> NOTES <tag> LineCount <tab> Prompt
First Line
Second Line
...
Last Line
```

USAGE:

```
Notes.Printl(Tag)
```

Tag STRING Optional. Used to print a different Tag than the one used in the call to .New

## Notes.Value()



### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Copy the Notes text to a string variable. This is limited in length to the first 255 characters.

USAGE:

```
Text = Notes.Value()
```

Text STRING

## NOTES.Dialog()



### *Related Topics*

TYPE: LICENSE

PURPOSE: To allow Notes to be edited in a full view dialog box.

Setup() will display a push button next to the reduced notes entry area.

When this button is pushed, an edit box will open with a larger view of the Notes text displayed. This text may be edited or cleared.

## Notify()



### *Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Write a short string in the left most region of the Runner window status bar. The string is generally used to notify the operator when the interpreter script begins to execute a new section of the experiment

USAGE:

Notify(Text)

Text STRING

## Notify2()



### *Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Write a short string in the right-most area of the Runner window status bar. The string is generally used to notify the operator about various script operations.

USAGE:

Notify2(Text)

Text STRING

## ObjectNew()



### *Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: To create a new instance of a Class Object.

USAGE:

self = ObjectNew(SomeClass)

self CLASS The new variable of CLASS SomeClass.

SomeClass TEXT The Class type specifier (Such as COMPLEX or CPIV)

## OCV



**Related Topics**

TYPE: CLASS

PURPOSE: The OCV class manages data acquisition for a Controlled Voltage I-V curve. However, it is a special purpose curve designed for measuring the open circuit voltage over time. The measurement is made in the Potentiostatic mode but with the Cell Switch open. The operator may set a voltage stability limit. When this limit is met the Ocv terminates.

## OCV.New()

**Related Topics**

TYPE: CLASS FUNCTION

PURPOSE: Create an object of the OCV class. Don't run the curve yet.

USAGE:

```
Ocv = OCV.New(Tag, Rate, Timeout)
```

Tag STRING Object Tag, used by Ocv.Printl()

Pstat Object Pstat is an object with a PSTAT license

Ocv OCV The curve object

## Ocv.StopAt()

**Related Topics**

TYPE: INSTANCE FUNCTION

PURPOSE: Stop the curve if either the rate of change in E is too small or too great. The first case is used to terminate a curve when the sample has stabilized. The second test is useful in detecting an unstable sample. Note that there is a default 10 point delay on any stop test due to the inherent Bessel Filter used in the OCV curve.

1. Stable  $d(E_{abs})/dt < dEdt_{min}$
2. Runaway  $d(E_{abs})/dt > dEdt_{max}$

USAGE:

```
Ocv.StopAt(Stable, Runaway)
```

Stable	REAL	Enable the Stable test and set value
	or NIL	Disable the Stable test
Runaway	REAL	Enable the Runaway test and set value
	or NIL	Disable the Runaway test

## Ocv.Run()

**Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Run the curve. When Run() is executed, the acquired data is displayed in a real-time updating curve. There are four buttons active during Cpiv.Run() - Abort, Pause/Continue, Skip and Point. The Abort button terminates the entire experiment. The Skip button skips to the end of the Ocv.Run and continues on with the next phase of the experiment.

Note: The Ocv.Run() function does not turn off the cell. The script must use the PStat.SetCell() function to control the cell.

**USAGE:**

```
Result = Ocv.Run()
```

Result INDEX 0 = Normal finish or Skip button pushed  
1 = Stop due to Stable Criterion  
2 = Stop due to Runaway Criterion

## Ocv.LastE()

**Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Find the last measured E value in an Ocv curve. This can be used as the measured Open Circuit Voltage by statements like:

```
Ocv.Run()  
POTEN.SetEoc(Ocv.LastE())
```

You should check the results of Ocv.Run() to see if the last measured E is stable.

**USAGE:**

```
Eoc = Ocv.Run()
```

Eoc REAL Last measured voltage

## Ocv.Print()

**Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Ocv.Print() will append the contents of an Ocv object to the Output file. The data format is:

```
Tag <tab> "TABLE" <tab> Points  
<tab> "Pt" <tab> "T" <tab> "Vf" <tab> "Vm" <tab> "Over"  
<tab> "#" <tab> "s" <tab> "v" <tab> "v" <tab> "bits"  
<tab> Pt1 T V V Flags  
<tab> Pt2 T V V Flags  
...  
<tab> PtN T V V Flags
```

where the symbols have the following meanings

--	--

Tag	The Tag parameter from the call to OCV.New
Points	Total number of points actually acquired
Pt	Sequential number of a point.
T	Elapsed time from the start of the experiment
Vf	Filtered Voltage
Vm	Unfiltered Voltage
Over	Overload flags

**USAGE:**

Ocv.Printl(Tag)

Tag STRING Optional. Used to print a different Tag than the one used in the call to .New

## Ocv.SetPlot()



*Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Set the form of the Ocv real-time plot. Plots may be shown as filtered V versus Time, unfiltered V versus Time or a dual filtered/unfiltered V versus time. The recommended format is filtered V versus time.

**USAGE:**

Ocv.SetPlot(View, MinFV, MinUV, MinT)

View INDEX A code indicating what is to be displayed. Use the following constants:

*OCV\_VTSMOOTH* = unfiltered V

*OCV\_VTRAW* = filtered V

*OCV\_VTDUAL* = both filtered and unfiltered V

MinFV Real Minimum filtered voltage value to display (optional)

MinUV Real Minimum unfiltered voltage value to display (optional)

MinT Real Minimum time value to display (optional)

## Ocv.SetPlotView()



*Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Set the number of graphs displayed in the real time display.

**USAGE:**

Ocv.SetPlotView(Format)

Format INDEX A code indicating the number of graphs displayed

Use the following constants:

*VIEW\_NONE*

*VIEW\_SINGLE*

*VIEW\_DOUBLE*

## Ocv.SetAxis()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Helps to define the real time display of an OCV object. Associates an axis with a data column in the object. Controls linear/log display formatting and allows labeling of the axis.

You must call Ocv.SetAxis for each of the axes in the real time plot. For a Y and Y' versus X plot you need 3 calls to Ocv.SetAxis().

**USAGE:**

Ocv.SetAxis(AxisNo, Source, Scale, MinVal, Label, Units)

AxisNo	INDEX	Use the constants below to define which axis is being configured.
		<i>X_AXIS</i> X axis
		<i>Y_AXIS</i> Y axis (Y vs X plot) or Lower Y (Y, Y' vs X)
		<i>Z_AXIS</i> Upper Y axis (Y, Y' vs X only)
Source	INDEX	The column number containing the data to be plotted or use the predefined constants defined in <a href="#">Ocv.DataCol()</a> .
Scale	INDEX	<i>LIN_AXIS</i> <i>The axis is linear</i>
		<i>LOG_AXIS</i> <i>The axis is log</i>
MinVal	REAL	Minimum value used in prevention of "ugly" plot scaling. MinVal is used differently depending on Scale setting.
		On a linear axis, this is the minimum resolution of the plot. The range of values shown on the axis cannot be smaller than MinVal. This prevents excessively fine scaling when plotting data where all the values are identical or very similar.
		On a log axis, this value is substituted for 0's in the data. This prevents infinite log values.
Label	STRING	The label used for the axis. Short names (one or two character labels are preferred).
Units	STRING	The units used for the axis. Short names (one or two character labels are preferred).

## Ocv.Sprint()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Print the status of the Ocv to a string. The object's tag and last acquired data point will be printed in the format:

Tag,PointNumber

**USAGE:**

Result = Ocv.Sprint()

Result STRING Resulting tag & point number (0 based).

## Ocv.SprintPoint()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Used in output of data from a OCV object. A data point from the object is printed to an ASCII string. This string can in turn be printed to the active output file.

The format of the data string is:

PointNumber <tab> Time <tab> Vf <tab> Vm <tab> Over

You can use this function along with the Printl() function to output point i to the output file as follows:

```
Printl("\t",Ocv.SprintPoint(i))
```

**USAGE:**

Result = Ocv.SprintPoint(PointNumber)

Result STRING The output string

PointNumber INDEX The point to be printed. PointNumber is 0 based. If the requested point has not been taken, a parameter error will be issued.

Time REAL Time from start of experiment in seconds

Vf REAL Filtered Voltage

Vm REAL Unfiltered Voltage

Over STRING Hexadecimal number representing overload status

## Ocv.DataValue()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Extract a single data item from a OCV object. The data item is taken from the ith data point and the jth data column. If no data has been taken the returned value is NIL.

**USAGE:**

Data Value = Ocv.DataValue(PointNum,ColNumber)

Result REAL The returned value.  
or  
NIL No data acquired yet.

PointNum INDEX Point number, zero based. If PointNum is out of range, a value from the last or acquired point is returned.

NIL Always return a value from the last acquired point.

ColNumber INDEX Data column, with a range of 0 to 2. The meaning of the data in each column is given in the [Ocv.DataCol\(\)](#) description.

## Ocv.DataCol()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Create and return an object of the class DATACOL. The object is filled with data extracted from one column of the Ocv object's data array. The data, once extracted, can then be submitted to other objects for further calculation. For example, a LINFIT object performs a linear regression analysis on two DATACOL objects.

A new DATACOL object is automatically created by the call to Ocv.DataCol(). It is usually transient, existing only for one function call.

**USAGE:**

```
DataCol = Ocv.DataCol(Column)
```

DataCol DATACOL The new object containing the data extracted from the Ocv data array.

Column INDEX A code identifying the column to be used. The following predefined constants should be used:  
*OCV\_T* = Time  
*OCV\_Vf* = Filtered Voltage  
*OCV\_Vm* = Unfiltered Voltage

## Ocv.Count()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Return the number of points actually acquired in a OCV object. The number returned is not the number of points allocated.

**USAGE:**

```
NumPoints = Ocv.Count()
```

NumPoints INDEX The number of points in the OCV object. Zero is returned if no data has been acquired yet.

## Ocv.Activate()



### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Make the real time plot of the Ocv object the Active plot. If the Runner window Curve List control is set to Active, the plot will be displayed. If the Runner window Curve List control is set to a definite curve, the display does not change.

USAGE:

Ocv.Activate()

## ONEPARAM



### *Related Topics*

TYPE: CLASS

PURPOSE: The ONEPARAM class allows the creation of a complex object consisting of one BOOL and one REAL bundled into one object. The advantage of bundling is that a Oneparam uses only one line in SETUP().

## ONEPARAM.New()



### *Related Topics*

TYPE: CLASS FUNCTION

PURPOSE: Create and initialize a new Oneparam object.

USAGE:

Oneparam = ONEPARAM.New(Tag, Bval, Val1, Prompt, Prompt1)

Tag STRING Used by Setup() Save/Restore,  
Tag.Printl().

Bval BOOL Initial value for BOOL within Oneparam.

Val1 REAL Initial value for REAL.

Prompt STRING Prompt string for whole object.

Prompt1 STRING Prompt string for REAL.

Oneparam ONEPARAM The initialized object.

## Oneparam.Printl()

**Related Topics**

TYPE: INSTANCE FUNCTION

PURPOSE: Print a Oneparam to the current Output file. The Oneparam will be printed in the format:

Tag <tab> ONEPARAM <tab> BOOL <tab> REAL1 <tab> Prompt <tab> Prompt1

USAGE:

Oneparam.Printl(Tag)

Tag STRING Optional. Prints a different Tag than the one used in the call to .New

**Oneparam.V1()****Related Topics**

TYPE: INSTANCE FUNCTION

PURPOSE: Copy the current value of the REAL embedded in a Oneparam into a REAL.

USAGE:

Value = Oneparam.V1()

Value REAL Current value of the Oneparam's REAL

**Oneparam.SetV1()****Related Topics**

TYPE: INSTANCE FUNCTION

PURPOSE: Set the current value of the REAL embedded in a Oneparam. Returns the value which was just set.

USAGE:

Oneparam.SetV1(Value)

Value REAL Value to set the Oneparam's REAL

**Oneparam.Check()****Related Topics**

TYPE: INSTANCE FUNCTION

PURPOSE: Copy the current value of the BOOL embedded in a Oneparam into a BOOL.

USAGE:

Value = Oneparam.Check()



Value **BOOL** Current value of the Oneparam's **BOOL**.

## ONEPARAM.Dialog



*Related Topics*

TYPE: LICENSE

PURPOSE: Allow a Oneparam to be edited, saved, and restored in Setup(). The Oneparam will be edited using the fields:

Prompt [X] On(or Off) Prompt1 Val1

Selecting the checkbox changes the state of the **BOOL**, changes the label after the checkbox, and grays or ungrays the rest of the object's additional prompts and values. When the checkbox shows a check mark, the **BOOL** is **TRUE**, the label is "On" and the text is black. When the checkbox is not checked, the **BOOL** is **False**, the label is Off, and the text is grayed out and cannot be edited.

When the operator enters a value into Val1, Setup will attempt to turn it into a valid **REAL** and replace the **REAL** in Oneparam with it. For example, if an operator puts in "1", Setup() will change it to "1.0". If Setup() cannot translate a string into a **REAL**, it will default to the value 0.0.

## OUTPUT



*Related Topics*

TYPE: CLASS

PURPOSE: The **OUTPUT** class is used for file management. Each Output object describes a file which may be open or closed. There can only be one open Output for each running script. This is the file to which all Print and Printl statements will write. You can, however, close one Output file and open another.

If the Disk is full on any function calls that write to the Output object, the following occurs:

- A message box asking for a new output filename and disk drive is displayed.
- If the operator responds within 60 seconds, the system proceeds with the new filename.
- If the operator does not respond in 60 seconds, the system will attempt to create and write to a temporary file named "CMS####.TMP" in the current subdirectory of drive C:. #### is a unique number. This is done assuming that the full disk is a floppy diskette. If this fails, the data is lost.

## OUTPUT.New()



*Related Topics*

TYPE: CLASS FUNCTION

PURPOSE: To create and initialize (but not open) a new Output object. For example:

```
Output = OUTPUT.New("OUTFILE", "Expout.dta", "Results File")
```

USAGE:

```
Output = OUTPUT.New(Tag, Filename, Prompt)
```

Tag **STRING** Object Tag used in Setup(), Tag.Printl()

Filename STRING File pathname (max of 80 char)

Prompt STRING Prompt text used in Setup()

Output OUTPUT The output object.

## Output.FileName()



TYPE: INSTANCE FUNCTION

PURPOSE: Extract the file name from an Output into a string.

USAGE:

```
FileName = Output.FileName()
```

FileName STRING

## Output.Open()



TYPE: INSTANCE FUNCTION

PURPOSE: Open the Output file. Make it the current output file for the running experiment. Close any other open Output.

If a file of the same name already exists, optionally ask the operator if they want that file replaced. If they respond negatively, don't open output file and return FALSE. If the file cannot be opened, also return FALSE.

The runner window caption and window title bar is updated with the new output name.

USAGE:

```
Status = Output.Open(Force)
```

Status BOOL TRUE if file opened OK, FALSE if not OK

Force BOOL TRUE - always overwrite file without asking operator.  
FALSE or missing - ask operator if file to be overwritten

## Output.Append()



TYPE: INSTANCE FUNCTION

PURPOSE: Reopen an existing Output file and move the file pointer to the end of the file. Make it the current output file for the running experiment. Close any other open Output.

If the file does not already exist it is created.

The runner window caption and window title bar is updated with the new output name.

USAGE:

```
Status = Output.Append()
```

Status BOOL TRUE if file opened OK, FALSE if not OK

## Output.SetCommit()



TYPE: INSTANCE FUNCTION

PURPOSE: Changes the way in which the output file is written. The output file can be set up to update the DOS file directory after each write to the file or to wait until the file is closed before updating.

If the update occurs after each Printl() function executes, the disk will see a lot of activity. This will generate noise, flashing lights, and may slow other Windows applications that are running concurrently. However, the data file will be relatively safe from unexpected system crashes. If the system does fail for some reason, the data written to disk up to that point will be saved.

If the update is delayed until the file is closed, the system will operate faster and with less disk activity. However, if the computer system fails before the end of a run, the data file will be lost. The default setting is to delay update.

USAGE:

```
Output.SetCommit(Switch)
```

Switch BOOL TRUE for update after each write  
FALSE for update only upon file close  
(default setting)

## Output.Close()



TYPE: INSTANCE FUNCTION

PURPOSE: Close the Output file, remove its name from the caption. If a running experiment has no open Output, any output from Print and Printl statements will be lost.

USAGE:

```
Output.Close()
```

## OUTPUT.Dialog



TYPE: LICENSE

PURPOSE: Allow Output file name to be edited, saved, and restored in the Setup dialog box.

Setup will display an Output object as:

Prompt [Filename ]

where the Filename field is editable. Setup will save and restore Output using the Tag text as an index.

## Pause()



### *Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Pause the execution of this script for a specific number of seconds or until the user hits the **SKIP** button. This function will return control to Windows so that it may continue to process other windows and applications. If you have experiments running in other windows, they will continue to run normally.

USAGE:

Pause(Time)

Time INDEX Pause execution for Time seconds

## Phase()



### *Related Topics*

TYPE: Regular Function

PURPOSE: Calculate the phase in degrees of a complex number using  $\text{Phase} = \arctan(\text{im}/\text{re})$ .

USAGE:

Result = Phase(RealPart, ImagPart)

RealPart REAL Real part of a complex number.

ImagPart REAL Imaginary part of a complex number.

Result REAL Result of calculation in degrees.

## POTEN



### *Related Topics*

TYPE: CLASS

PURPOSE: The POTEN class creates a Voltage value that may refer to either Working Electrode vs Reference Electrode Voltage (vs Eref) or Working Electrode vs W.E. at Open Circuit Voltage (vs Eoc). The object data consists of two separate pieces of information, the voltage and the measurement mode (vs Eref or vs Eoc).

The Open Circuit Voltage (Eoc) is a number that is applied to all Poten's created from the class POTEN. When a Poten's value is used, if it's flag is in the vs Eoc state, the Eoc is added to the value to get the required voltage. If it's flag is in the vs Eref state, the value is used directly as the required voltage.

Therefore an operator can use a POTEN to specify either a voltage deviation from open circuit or an absolute voltage (vs ref). To get the one form from the other, use the formulae:

$E_{vsEref} = P_{working} - P_{ref}$

$E_{oc} = P_{working}(@I=0) - Pref$

$E_{vsEoc} = E_{vsEref} - E_{oc}$

where P's are potentials versus circuit ground.

## POTEN.New()



### *Related Topics*

TYPE: CLASS FUNCTION

PURPOSE: Create and initialize a new Poten object.

USAGE:

Poten = POTEN.New(Tag, Quantity, vsEoc, Prompt)

Tag STRING Used by Setup() Save/Restore,  
Poten.Printl().

Quantity REAL Input voltage.

vsEoc BOOL vs Eoc or vs Eref measurement  
TRUE = vs Eoc, FALSE = vs Eref

Prompt STRING Used by Setup() Edit for Operator  
Prompt.

Poten POTEN Object created.

## POTEN.SetEoc()



### *Related Topics*

TYPE: CLASS FUNCTION

PURPOSE: Set the Poten class's Eoc value. This value is usually acquired by measuring the potentiostat cell voltage with the cell switch off. Note that this value is applied to all objects of the class POTEN.

USAGE:

POTEN.SetEoc(Eoc)

Eoc REAL New voltage. Applied to all Poten  
objects.

or

NIL Set Eoc to 0.0

**POTEN.Eoc()***Related Topics*

TYPE: CLASS FUNCTION

PURPOSE: Read the Poten class's Eoc value.

USAGE:

```
Eoc = POTEN.Eoc()
```

Eoc REAL New voltage. Applied to all Poten objects.

**Poten.Print()***Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Print a Poten to the current Output file. The Poten will be printed in the format:

```
Tag <tab> POTEN <tab> Quantity <tab> T <tab> Prompt
```

or

```
Tag <tab> POTEN <tab> Quantity <tab> F <tab> Prompt
```

The fourth field is a BOOL with T representing the vs Eoc state and F representing the vs Eref state.

USAGE:

```
Poten.Print(Tag)
```

Tag STRING Optional. Used to print a different Tag than the one used in the call to .New

**Poten.VsEref()***Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Report the Poten's value versus Eref.

USAGE:

```
Voltage = Poten.VsEref()
```

Voltage REAL Calculate actual EvsEref using equations. If vsEoc switch is set, add

Eoc to quantity before returning  
Voltage

### Poten.SetValue()



TYPE: INSTANCE FUNCTION

PURPOSE: Set the potential value from the script.

USAGE:

Poten.SetValue(Quantity, VsEoc)

Quantity REAL New value of potential.  
NIL If NIL, value not changed

VsEoc BOOL T = Value is taken vs. Eoc. F = Value is  
taken vs. Eref.  
NIL Flag not changed.

### Poten.Sprint()



TYPE: INSTANCE FUNCTION

PURPOSE: Print the value of a Poten to a string.

USAGE:

Result = Poten.Sprint(Format)

Format STRING (OPTIONAL) C-language format string.  
Default is %G,%c  
with last field being either 'T' or 'F'.

Result STRING Resulting formatted string.

### Poten.Value()



TYPE: INSTANCE FUNCTION

PURPOSE: Report the Poten's value versus Eref. Same as Poten.VsEref, but added to increase readability of scripts.

USAGE:

Voltage = Poten.Value()

Voltage REAL Calculate actual EvsEref using equations. If vsEoc switch is set, add Eoc to quantity before returning Voltage

## POTEN.Dialog



*Related Topics*

TYPE: LICENSE

PURPOSE: Allow Poten to be edited, saved, and restored in Setup(). The Quant will be edited using the fields:

Prompt [Quantity ] [ ] vsEref

or

Prompt [Quantity ] [X] vsEoc

Setup treats the Quantity field similar to a Quant object and it treats the vsExxx portion similar to a Toggle object.

## Pow()



*Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Calculate a number raised to a power,  $x^y$ .

USAGE:

Result = Pow(X,Y)

X REAL Base value. If  $X < 0$ , -X is used.

Y REAL Exponent.

Result REAL Result of the calculation.

## Print()



*Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Print one or more values to the output file. Only the primitive data types (REAL, INDEX, STRING, BOOL, and BITS) will print usable values. For example:

```
ExpNo = 5
Print("Experiment No.", ExpNo)
```

prints

Experiment No.5



in the output file.

USAGE:

Print(Value, ...)

Value any type The value(s) to be printed

Each variable is printed according to its type as shown in the following table.

Variable	Prints
BOOL	"T" or "F"
INDEX	Integer value. No leading zeros. Negative sign if required.
REAL	General format. E notation for very large or small numbers.
BITS	Hexadecimal preceded by 0x. No leading zeros.
STRING	String value up to terminating null character.
NIL	"NIL"

## Printl()



### Related Topics

TYPE: REGULAR FUNCTION

PURPOSE: Same as Print() but Printl() puts an end of line character after the last value printed. See function Print() for details.

USAGE:

Printl(Value, ...)

## PstatActiveBitmap()



### Related Topics

TYPE: REGULAR FUNCTION

PURPOSE: Return the active potentiostats as BITS so a selector object can be formatted appropriately. This function is used in scripts which require a potentiostat. It allows users to know which potentiostats are currently in use. See the [Selector.SetStyle](#) discussion for more information on setting the style of a selector.

USAGE:

Pattern = PstatActiveBitmap()

Pattern BITS The current potentiostats already in use  
by the Framework.

## PSTAT



### Related Topics

TYPE: CLASS

PURPOSE: The PSTAT class encapsulates most of the features of a Gamry Instrument's potentiostat/galvanostat. It is used to provide control over a potentiostat and to keep two experiments from trying to share the same potentiostat.

**USAGE:** The PSTAT class is not actually used in practice. Rather, it is a phantom class that is superceded by the actual class of potentiostat, e.g. PCI4 or PC5. Each of the instance functions described in this help would actually pertain to a PCI4 or PC5 class, and not a phantom PSTAT class.

## PSTAT.New()



### Related Topics

**TYPE:** CLASS FUNCTION

**PURPOSE:** Create a new Pstat object. One can't actually create potentiostat hardware in software so the New function is used to identify which of the available hardware potentiostats will be used in the experiment. One should note that there actually is no Class PSTAT in this version of the Gamry Framework. PSTAT is a name which is used in place of PCI4 or PC5. The actual classes that are used by the Explain scripts are PCI4 and PC5, but they are never explicitly referenced. Rather, the classname of a potentiostat is read from the GAMRY.INI file and is used when creating the potentiostat.

**USAGE:**

```
Pstat = PSTAT.New(Tag, Section)
```

Tag STRING Used by Setup() Save/Restore,  
Pstat.PrintI()

Section STRING Used to determine which section in the  
GAMRY.INI contains the potentiostat information.

Pstat PSTAT Pstat object used by other class  
functions

## Pstat.ACCouple()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Report the current AC Coupling state.

**USAGE:**

```
Result = Pstat.ACCouple()
```

Result BOOL

TRUE DC Coupled

FALSE AC coupled

## Pstat.SetACCouple()

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** AC couple the Pstat input. The input will only be AC coupled if a) the Pstat is capable of AC coupling at all and b) the Pstat can AC couple signals at the chosen frequency.

**USAGE:**

```
Result = Pstat.SetACCouple(Switch, Frequency)
```

Switch	BOOL	Requested AC Coupling state. FALSE = DC Coupled, TRUE = AC coupled.
Frequency	REAL	The frequency that will be used.
Result	BOOL	The AC Coupling state after any change. FALSE = DC Coupled, TRUE = AC coupled.

This value should be checked. Some Pstat's will always return FALSE. All other Pstat's will have a frequency below which AC coupling does not work.

[INSTRUMENTS:](#) PCI4

## Pstat.AnalogOut()



*Related Topics*

[TYPE:](#) INSTANCE FUNCTION

[PURPOSE:](#) Read the current voltage setting for the auxiliary DAC output on the Miscellaneous I/O Connector.

[USAGE:](#)

```
VoltageSet = Pstat.AnalogOut()
VoltageSet    REAL    The current voltage as set.
```

## Pstat.SetAnalogOut()

[TYPE:](#) INSTANCE FUNCTION

[PURPOSE:](#) Set the voltage the auxiliary DAC output on the Miscellaneous I/O Connector.

[USAGE:](#)

```
VoltageSet = Pstat.SetAnalogOut(Voltage)
Voltage    REAL    The new output voltage.
VoltageSet REAL    The current voltage as set.
```



Note: For the PCI4 family of Potentiostats, the offset and/or full scale range of the DAC can be changed using jumpers on the Controller Card. This function automatically adjusts the DAC scaling using any appropriate scaling values found in the "GAMRY5.INI" file. See the Hardware Operator's Manual for further information.

[INSTRUMENTS:](#) PC5, PCI4

## Pstat.AchFilter()



*Related Topics*

[TYPE:](#) INSTANCE FUNCTION

**PURPOSE:** Report the current AchFilter setting.

**USAGE:**

Filter = Pstat.SetAchFilter()

Filter	INDEX	Filter to select
		0 = No Filter
		1 = 200 kHz
		2 = 1 kHz
		3 = 5 Hz
	or	
	BOOL	FALSE = Ground. No Pass Filter setting.

**INSTRUMENTS:** PC5

**Pstat.SetAchFilter()**



*Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Pstat.SetAchFilter() sets the 2 pole Butterworth low pass filter on the auxillary measuring channel.

If the parameter passed by SetAchFilter is a frequency in Hz, the driver will then select the most appropriate filter setting able to pass information corresponding to this frequency.

**NOTE:** The No Filter setting (0) should not be used under normal circumstances. It is to be used exclusively for AC impedance measurements beyond 100 kHz.

**USAGE:**

FilterSet = Pstat.SetAchFilter(Freq)

Freq	REAL	Frequency of interest in Hertz
	or	
	INDEX	Filter to select
		0 = No Filter
		1 = 200 kHz
		2 = 1 kHz
		3 = 5 Hz
	or	
	BOOL	FALSE = Ground. No Pass Filter setting.
FilterSet	INDEX	The new filter setting

**INSTRUMENTS:** PC5

## Pstat.AchOffset()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Report the offset voltage of the current channel.

USAGE:

Result = Pstat.AchOffset()

Result	REAL	The offset voltage.
--------	------	---------------------

INSTRUMENTS: PC5

## Pstat.SetAchOffset()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Set the DC offset voltage in the current channel. The actual offset circuitry may be implemented in the potentiostat or in the FRA. Not all hardware configurations have this feature. The actual DC offset voltage is returned. If the AchOffsetEnable flag is set to FALSE, 0 will be returned. If the hardware doesn't support the DC offset, 0 will also be returned.

USAGE:

Result = Pstat.SetAchOffset(Level)

Level	REAL	Offset voltage level. Both current and voltage channel offsets are specified in terms of voltage. Current values can be converted to voltages by multiplying by the potentiostat effective I/E resistance (See Pstat.IEResistor).
Result	REAL	The active (after any change) offset voltage. This value should be checked since some configurations will always return 0. If the AchOffsetEnable flag is set to FALSE, 0 will be returned.

INSTRUMENTS: PC5

## Pstat.AchOffsetEnable()

**Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Returns whether an offset for the I channel is enabled or disabled.

**USAGE:**

```
Result = Pstat.AchOffsetEnable()
```

Result	BOOL	TRUE = Enabled FALSE = Disabled
--------	------	------------------------------------

**INSTRUMENTS:** PC5

## Pstat.SetAchOffsetEnable()

**Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Enable or Disable post potentiostat offset correction. This allows the DC component of a measured voltage signal to be removed. Removal of this DC component allows multiplication of the signal by a post gain factor (1x, 10x, 100x) while keeping this post gain signal within the voltage limit of the analog to digital converter (ADC).

**USAGE:**

```
Result= Pstat.SetAchOffsetEnable(State)
```

State	BOOL	TRUE = Enable FALSE = Disable
Result	BOOL	Returns State as Set

**INSTRUMENTS:** PC5

## Pstat.AchRange()

**Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Reads the range of the Auxiliary A/D input. Currently this function performs no valid use on the PCI4, as this potentiostat has only one Ach Range.

**USAGE:**

```
RangeSet = Pstat.AchRange ()
```

RangeSet	INDEX	Range of Aux Channel 0 = 30 mV 1 = 300 mV 2 = 3 V
----------	-------	--

**INSTRUMENTS:** PC5

## Pstat.SetAchRange()

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Sets the range of the Auxiliary A/D input. Currently this function performs no valid use on the PCI4 potentiostat.

**USAGE:**

```
RangeSet = Pstat.SetAchRange (Range)
```

Range	INDEX	Range of Aux Channel
		0 = 30 mV
		1 = 300 mV
		2 = 3 V

RangeSet	INDEX	Range of Aux Channel.
----------	-------	-----------------------

**INSTRUMENTS:** PC5

## Pstat.AchSelect()



*Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Report the current AchSelect setting.

**USAGE:**

```
Filter = Pstat.SetAchSelect()
```

Filter	INDEX	Input selected
		0 = BNC connector (Default)
		1 = Pstat (Not used currently)
		2 = Control Amp on Pstat
		3 = Thermocouple
		4 = Ground

**INSTRUMENTS:** PC5

## Pstat.SetAchSelect()



*Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Pstat.SetAchSelect specifies the input to be measured by the Auxiliary channel on the control board. The default setting is the BNC input on the control board.

**USAGE:**

```
AchSelectSet = Pstat.SetAchSelect (Input)
```

Input	INDEX	Input to select 0 = BNC connector (Default) 1 = Pstat (Not used currently) 2 = Control Amp on Pstat 3 = Thermocouple 4 = Ground
AchSelectSet	INDEX	The new input selected

[INSTRUMENTS:](#) PC5

## Pstat.CableId()



*Related Topics*

[TYPE:](#) INSTANCE FUNCTION

[PURPOSE:](#) Report the current ID of the cable connected to the potentiostat. For potentiostats with a single cell cable, like the Reference 600 and Interface 1000, there is only one cable ID. For a Potentiostat like the Reference 3000 with multiple cell cables, the CableType needs to be specified in order to read the Power or Aux Electrometer cable IDs.

[USAGE:](#)

CableId = Pstat.CableId()

Cable Type	INDEX	Cable Id to Read
	0	Main Cell Cable
	1	Power Cable
	2	Aux Electrometer Cable

or use constants

CID_MAIN	Main Cell Cable
CID_PWR	Power Cable
CID_AE	Aux Electrometer Cable

CableId	INDEX	ID of the Cable Attached
	15	No Cable Connected
	14	60 cm Shielded
	13	1.5 m Shielded
	12	3 m Shielded
	11	10 m Shielded
	10	1m Low Inductance
	9	1m ECM8 Interconnect
	8	Booster Interconnect
	7	60 cm Shielded (Fuses in Hood)
	1	Reserved for custom cables
	0	Reserved

[INSTRUMENTS:](#) PC5



## Pstat.SetCableId()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Set a custom cable Id to the potentiostat. In general, this function should not be used, because the CableId should be read from the cable. In certain circumstances, this function could be used to read out the calibration values for a cable even though it is not connected. Any subsequent calls to Pstat.CableId () will clear the manual setting and replace it with the CableId read from the cell cable.

**USAGE:**

```
CableIdSet = Pstat.SetCableId (CableId)
```

CableId	INDEX	Cable Id to Set (Valid Index from 0 to 15)
---------	-------	---

CableIdSet	INDEX	The new CableId setting
------------	-------	-------------------------

**INSTRUMENTS:** PC5

## Pstat.CalDate()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Report the current CalDate setting.

**USAGE:**

```
Date = Pstat.SetCalDate(Type)
```

Type	INDEX	Type of calibration to return date 0 = DC Calibration Date
------	-------	---

Date	STRING	1 = AC Calibration Date Date of last calibration
------	--------	---

**INSTRUMENTS:** PC5, PCI4

## Pstat.SetCalDate()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Pstat.SetCalDate is used to write the calibration date to either the GAMRY5.INI file for PCI4 family instruments, or to non-volatile memory in PC5 family instruments. In general, this function should only be used by the calibration scripts.

USAGE:

CalDateSet = Pstat.SetCalDate (Type, Date)

Type	INDEX	Type of calibration 0 = DC Calibration Date 1 = AC Calibration Date
Date	STRING	Date to write
CalDateSet	STRING	The value written

INSTRUMENTS: PC5, PCI4

## Pstat.CalibKey()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Report the current value of the specified Calibration Key. These values are used to correct for offsets in the potentiostat. If you attempt to read a Key which does not exist a run error will be issued.

USAGE:

Result = Pstat.CalibKey(Key)

Key	STRING	The Key of the calibration value to lookup.
Result	VECTOR	The current calibration values belonging to that key. The vector length will vary by Key.

INSTRUMENTS: PC5, PCI4

## Pstat.SetCalibKey()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Set the current value of the specified Calibration Key. These values are set in the calibration scripts and should not be modified. If you attempt to set a Key which does not exist a run error will be issued.

USAGE:

Pstat.SetCalibKey(Key, Vector)

Key	STRING	The Key of the calibration value to set.
Vector	VECTOR	A vector containing the calibration information

INSTRUMENTS: PC5, PCI4

## Pstat.CASpeed()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Report Control Amplifier Speed setting.

**USAGE:**

```
Switch = Pstat.CASpeed()
```

**INSTRUMENTS:** PC5, PCI4

## Pstat.SetCASpeed()

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Pstat.SetCASpeed() sets the roll off filter on the potentiostat control amp. It can be varied from 0 to 3 with 0 being the fastest speed.

The control amplifier roll off affects the overall stability of the potentiostat. As with the Pstat.SetStability() disclaimer, we can only offer you a guideline in setting the CASpeed.

If the potentiostat oscillates on all current ranges change the setting.

Four constants have been set to make code reading easier. Located in the pstat.exp script, their names are:

```
CASpeedFast
CASpeedNorm
CASpeedMed
CASpeedSlow
```

**USAGE:**

```
SwitchSet = Pstat.SetCASpeed (Switch)
```

Switch	INDEX	Allowed values with lower numbers being faster (Hardware dependent) 0 - 4 for PC5 0 - 3 for PCI4
--------	-------	---

or

SwitchSet	REAL	Bandwidth Required (Hertz) <i>Preferred!</i>
SwitchSet	INDEX	The new setting



**WARNING:** Oscillation can result with the two fastest CA Speeds on a PC5 (0,1) or the fastest speed (0) on a PCI4, when the control mode is ZRA. These CA Speeds should not be used in ZRA mode.

**INSTRUMENTS:** PC5, PCI4

## Pstat.Cell()

**Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Return the current state of the Cell Switch.

**USAGE:**

```
SwitchSet = Pstat.Cell()
```

SwitchSet

CellOff      No cell current can flow. The counter electrode is disconnected.

CellOn        Cell current may flow. The counter electrode is connected to the cell.

Since the Reference Electrode is always connected, measurements of the open circuit voltage can be made even if the Cell Switch is in the CellOff position.

**INSTRUMENTS:** PC5, PCI4

## Pstat.SetCell()

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Turn on/off the Cell Switch and return the current state of the Cell Switch.

**USAGE:**

```
SwitchSet = Pstat.SetCell(Switch)
```

Switch,  
SwitchSet

CellOff      No cell current can flow. The counter electrode is disconnected.

CellOn        Cell current may flow. The counter electrode is connected to the cell.

**INSTRUMENTS:** PC5, PCI4

## Pstat.Close()

**Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Free the potentiostat from this experiment. Allow it to be used in other experiments.

**USAGE:**

```
Pstat.Close()
```

**INSTRUMENTS:** PC5, PCI4

## Pstat.Convention()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Report current measurement convention

USAGE:

```
Switch = Pstat.Convention()
```

INSTRUMENTS: PC5, PCI4

## Pstat.SetConvention()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Set the current reporting convention to either Anodic = Positive or Cathodic = Positive.

USAGE:

```
SwitchSet = Pstat.SetConvention(Switch)
```

Switch INDEX 0 = Anodic Positive  
(Corrosion Convention)  
1 = Cathodic Positive  
(Voltammetry Convention)

SwitchSet INDEX The new current convention.

Two constants have been set to make code reading easier:

*Anodic* (0)

*Cathodic* (1)

## Pstat.CtrlMode()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Return the current potentiostat control mode.

USAGE:

```
ModeSet = Pstat.CtrlMode()
```

ModeSet

GstatMode Galvanostat, Control I

PstatMode Potentiostat, Control V  
 ZraMode Zero Resistance Ammeter  
 FraMode FRA

[INSTRUMENTS:](#) PC5, PCI4

## Pstat.SetCtrlMode()

[TYPE:](#) REGULAR FUNCTION

[PURPOSE:](#) Switch the potentiostat between Control V (Potentiostat) and Control I (Galvanostat) modes.

[USAGE:](#)

```
ModeSet = Pstat.SetCtrlMode(Mode)
Mode, ModeSet
GstatMode Galvanostat, Control I
PstatMode Potentiostat, Control V
ZraMode Zero Resistance Ammeter
FraMode FRA
```

[INSTRUMENTS:](#) PC5, PCI4

## Pstat.DDSAmpl()



*Related Topics*

[TYPE:](#) INSTANCE FUNCTION

[PURPOSE:](#) Reports the current RMS voltage of the DDS Sine wave synthesizer.

[USAGE:](#)

```
Result = Pstat.DDSAmpl ()

Result          REAL          Current RMS amplitude
```

## Pstat.SetDDSAmpl()



*Related Topics*

[TYPE:](#) INSTANCE FUNCTION

[PURPOSE:](#) Sets the RMS voltage of the DDS Sine wave synthesizer. For a sine wave  $V_{rms} = V_{peak} / \sqrt{2} = (V_{pk} \text{ to } pk) / (2 * \sqrt{2})$ . The MAX RMS voltage varies by potentiostat and is 3.62 V (5.12 V Peak) on a PCI4 or Series G family Potentiostat. PC5 family Potentiostats have a 1.448V RMS maximum (2.048V Peak). Requested voltages which are higher than the maximum value will be limited to the maximum value.

[USAGE:](#)

```
Result = Pstat.SetDDSAmpl (Ampl)

Ampl          REAL          Requested RMS amplitude (Volts)
```

Result	REAL	Resulting RMS amplitude as set
--------	------	--------------------------------

## Pstat.DDSEnable()



### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Report the current state of the DDS Sine wave synthesizer

USAGE:

State = Pstat.DDSEnable ()

State	BOOL	TRUE = Turn On FALSE = Turn Off
-------	------	------------------------------------

## Pstat.SetExtSrcEnable()



### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Turn the DDS Sine wave synthesizer on or off

USAGE:

Result = Pstat.SetDDSEnable (State)

State	BOOL	TRUE = Turn On FALSE = Turn Off
Result	BOOL	Return State as set

## Pstat.DDSFreq()



### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Reports the current frequency of the DDS Sine wave synthesizer

USAGE:

Freq = Pstat.DDSFreq ()

Freq	REAL	Current DDS Frequency (Hz)
------	------	----------------------------

## Pstat.SetDDSFreq()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Sets the frequency of the DDS Sine wave synthesizer. The upper and lower limits vary based on the instrument.

**USAGE:**

Result = Pstat.SetDDSFreq (Freq)

Freq	REAL	Requested Frequency (Hz)
Result	REAL	Resulting frequency as set

## Pstat.DigitalIn()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Read the digital input bits on the potentiostat's Miscellaneous I/O Connector.

There are 4 uncommitted digital inputs in the Miscellaneous I/O Connector. The bits are labeled In0, In1, In2 and In3. See your potentiostat's Operator's Manual for the pin assignments in the Miscellaneous I/O Connector.

**USAGE:**

Reading = Pstat.DigitalIn()

Reading BITS The input code. In0 is the LSB.

## Pstat.DigitalOut()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Read the current digital output setting.

See Pstat.SetDigitalOutput() for more information.

**USAGE:**

Setting = Pstat.DigitalOut()

Setting BITS The current Output setting in the 4 least significant bits.

## Pstat.SetDigitalOut()

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Control external devices using the digital outputs available on the potentiostat's Miscellaneous I/O Connector.

There are 4 bits of uncommitted digital output on the Miscellaneous I/O Connector. These bits are high speed CMOS outputs (74HCTxxx family). A 330  $\Omega$  resistor in series with the output is used to protect against accidental short circuits.



Individual bits can be set or all 4 bits can be controlled simultaneously. The bits are labeled Out0, Out1, Out2 and Out3. See your potentiostat's Operator's Manual for the pin assignments in the Miscellaneous I/O Connector.

There are two different ways to call this function. In the first syntax, BITS arguments are used to show the desired bit pattern and to show the range of bits to be changed. Remember that a BITS constant has the form 0xXXXX with X representing any hexadecimal digit.

Each of the lowest 4 bits of the BITS argument corresponds to one of the output bits. A bit will only be changed if the mask argument has a one in that bit's bit position. A mask argument of 0x0003 would allow changes in Output0 and Output1. With this mask value, Output 2 and Output 3 will not be changed regardless of the bit pattern argument.

In the second calling syntax 4 BOOL parameters list the state of each bit. This format may be easier for the average user to understand.

#### USAGE:

Pstat.SetDigitalOut(Setting, Mask)

Setting BITS New output code in the lowest 4 bits.

A one in this argument causes a logic high at the connector.

Mask BITS Identify the bits to be changed.

or

Pstat.SetDigitalOut(Bit0, Bit1, Bit2, Bit3)

BitN BOOL The bit setting.

A TRUE causes the output to be a logic one.

A FALSE causes the output bit to be a logic zero.

or

NIL Don't change this bit.

## Pstat.ExtSrceEnable()



### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Report the current state of the External Source Input to the Potentiostat

#### USAGE:

State = Pstat.ExtSrceEnable ()

State	BOOL	TRUE = Turn On
		FALSE = Turn Off

## Pstat.SetExtSrceEnable()



### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Turn the External Source Input to the Potentiostat on or off

USAGE:

```
Result = Pstat.SeExtSrceEnable (State)
```

State	BOOL	TRUE = Turn On FALSE = Turn Off
Result	BOOL	Return State as set

## Pstat.FindIchRange()

*Related Topics*TYPE: INSTANCE FUNCTIONPURPOSE: FindIchRange() will acquire several current points, adjusting the current range after each point is taken. You use it prior to starting a scan, to home in on the optimum current range to use during the scan.

FindIchRange() uses the currently active settings for the potential, cell switch, filters and IR correction mode. The SKIP and ABORT buttons are active during FindIchRange().

USAGE:

```
Pstat.FindIchRange()
```

## Pstat.FindIERange()

*Related Topics*TYPE: INSTANCE FUNCTIONPURPOSE: FindIERange() will acquire several current points, adjusting the current range after each point is taken. You use it prior to starting a scan, to home in on the optimum current range to use during the scan.USAGE:

```
Pstat.FindIERange()
```

## Pstat.FindVchRange()

*Related Topics*TYPE: INSTANCE FUNCTIONPURPOSE: FindVchRange() will acquire several voltage points, adjusting the voltage measurement range after each point is taken. You use it prior to starting a scan, to home in on the optimum voltage range to use during the scan.

FindVchRange() uses the currently active setting for the current, cell switch, filters and IR correction mode. The SKIP and ABORT buttons are active during FindVchRange().

USAGE:

```
PStat.FindVchRange()
```

## Pstat.FreqLimitLower()

**Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Report the Pstat's minimum allowed frequency. Each driver reports the limitation imposed by its associated hardware.

**USAGE:**

```
Result = Pstat.FreqLimitLower()
```

Result	REAL	Minimum frequency.
--------	------	--------------------

**Pstat.FreqLimitUpper()****Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Report the Pstat's maximum allowed frequency. Each driver reports the limitation imposed by its associated hardware.

**USAGE:**

```
Result = Pstat.FreqLimitUpper()
```

Result	REAL	Maximum frequency.
--------	------	--------------------

**Pstat.Ground()****Related Topics**

**TYPE:** INSTANCE FUNCTION (PCI4 FAMILY ONLY!)

**PURPOSE:** Read the current status of the ground isolation switch.

**USAGE:**

```
State = Pstat.Ground()
```

State

Float Potentiostat floating ground is isolated from Earth Ground

Earth Potentiostat floating ground is connected to Earth Ground



Connection of the floating ground to the chassis precludes use of the potentiostat with cells that have an earthed electrode.

**Pstat.SetGround()**


**TYPE:** INSTANCE FUNCTION (PCI4 FAMILY ONLY!)

**PURPOSE:** Connect/disconnect the potentiostat's floating ground to/from Earth ground. Earth grounding the potentiostat can lower noise on cells that are isolated from earth ground.

**USAGE:**

```
Pstat.SetGround(Switch)
```

- Switch
- Float     Potentiostat floating ground is isolated from Earth Ground
- Earth     Potentiostat floating ground is connected to Earth Ground



This function causes no effect on the Reference 600. Consult the Reference 600's Hardware Operator's Manual for information about Earth Grounding this instrument.

### Pstat.GstatRatio()



TYPE: INSTANCE FUNCTION

PURPOSE: Returns the number of amps/input volt generated when in galvanostat mode. It is sensitive to sign convention (See Pstat.SetConvention) since amps are defined differently by convention. It is used in calibration but it can also be used to configure external signal sources.

USAGE:

Result = Pstat.GstatRatio(IERange)

IERange INDEX Current Range

Result REAL Amps/input volt in galvanostat mode.

### Pstat.IERange()



TYPE: INSTANCE FUNCTION

PURPOSE: Read the current range. Not all ranges are available on specific potentiostats. Please consult your potentiostat operator's manual for information specific to your potentiostat.

NOTE: The ranges listed below are for 300mA or 30 mA models. For 750 mA models, multiply the ranges by 2.5. For 600 mA models, multiply the ranges by 2.0.

USAGE:

Range =  
Pstat.IERange ()

Range	INDEX	Absolute current range (Full Scale Limit)
		0 = 3 pA
		1 = 30 pA
		2 = 300 pA
		3 = 3 nA

- 4 = 30 nA
- 5 = 300 nA
- 6 = 3 uA
- 7 = 30 uA
- 8 = 300 uA
- 9 = 3 mA
- 10 = 30 mA
- 11 = 300 mA
- 12 = 3 A
- 13 = 30 A
- 14 = 300 A
- 15 = 3 kA

**Pstat.SetIERange()**



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Set the current range. You can either set an absolute current range or set current range to a range large enough to measure a given current.

Not all ranges are available on specific potentiostats. Please consult your potentiostat operator’s manual for information specific to your potentiostat.

NOTE: The ranges listed below are for 300mA or 30 mA models. For 750 mA models, multiply the ranges by 2.5. For 600 mA models, multiply the ranges by 2.0.

USAGE:

RangeSet = Pstat.SetIERange(Range)

Range	INDEX	Absolute current range (Full Scale Limit)
		0 = 3 pA
		1 = 30 pA
		2 = 300 pA
		3 = 3 nA
		4 = 30 nA
		5 = 300 nA
		6 = 3 uA
		7 = 30 uA
		8 = 300 uA
		9 = 3 mA
		10 = 30 mA
		11 = 300 mA
		12 = 3 A
		13 = 30 A
		14 = 300 A
		15 = 3 kA

If you attempt to set a range which is not available for a specific potentiostat, a Parameter Error will be issued.

Range	or REAL	Current to be measured. Function picks best range.
-------	------------	--

If you attempt to set the range to measure a current greater than I<sub>max</sub> for a specific potentiostat, a Parameter Error will be issued.

RangeSet	INDEX	The new current range setting
----------	-------	-------------------------------

### Pstat.IERangeLowerLimit()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Reads the lower limit of the IERange as currently set.

USAGE:

Result = Pstat.IERangeLowerLimit ()

Result	INDEX	Lower Limit as Set
--------	-------	--------------------

### Pstat.SetIERangeLowerLimit()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Set the lower limit of the IERange. This limit is used when the potentiostat is making autorange decisions. By setting this lower limit, one can insure that the potentiostat will never range to a range lower than that specified by the limit. This is helpful when doing measurements at high speed.

USAGE:

Result = Pstat.SetIERangeLowerLimit (IERange)

Result	INDEX	Lower Limit as Set
--------	-------	--------------------

IERange	INDEX	The IERange which is to be used as the lower limit by the potentiostat
	NIL	Clear the IERange lower limit, and use the physical limit of the potentiostat

### Pstat.IERangeMode()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Return the current status of the autoranging mode of the I/E converter.

USAGE:

Mode = Pstat.IERangeMode()

Mode BOOL TRUE is autoranging to be enabled  
(default setting)  
FALSE if autoranging to be disabled.

### Pstat.SetIERangeMode()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Enable or disable current measurement autoranging.

If autoranging is enabled (on), after each current measurement is made, the current range is adjusted before the next point is recorded. The new current range is calculated so that the measured current will optimally fill the A/D converter with little danger of overflow.

By default current autoranging is enabled. Current autoranging is not recommended if data is being acquired faster than 1 point per second (with default filter settings).

**USAGE:**

```
Pstat.SetIERangeMode(Mode)
```

Mode BOOL TRUE is autoranging to be enabled  
(default setting)  
FALSE if autoranging to be disabled.

## Pstat.IchFilter()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Report the current IchFilter setting.

**USAGE:**

```
Filter = Pstat.SetIchFilter()
```

Filter	INDEX	Filter to select
		0 = No Filter
		1 = 200 kHz
		2 = 1 kHz
		3 = 5 Hz
	or	
	BOOL	FALSE = Ground. No Pass Filter setting.

## Pstat.SetIchFilter()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Pstat.SetIchFilter() sets the 2 pole Butterworth low pass filter on the current measuring channel. This is a different filter than the I/E Filter described in the SetStability() section. It has no effect on potentiostat stability, only on the current measuring circuit's frequency response.

If the parameter passed by SetIchFilter is a frequency in Hz, the potentiostat will then select the most appropriate filter setting able

to pass information corresponding to this frequency.

NOTE: The No Filter setting (0) should not be used under normal circumstances. It is to be used exclusively for AC impedance measurements beyond 100 kHz.

USAGE:

FilterSet = Pstat.SetIchFilter(Freq)

Freq	REAL or INDEX	Frequency of interest in Hertz  Filter to select  0 = No Filter  1 = 200 kHz  2 = 1 kHz  3 = 5 Hz
	or BOOL	FALSE = Ground. No Pass Filter setting.
FilterSet	INDEX	The new filter setting

**Pstat.IchOffset()**



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Report the offset voltage of the current channel.

USAGE:

Result = Pstat.IchOffset()

Result	REAL	The offset voltage.
--------	------	---------------------

**Pstat.SetIchOffset()**



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Set the DC offset voltage in the current channel. The actual offset circuitry may be implemented in the potentiostat or in the FRA. Not all hardware configurations have this feature. The actual DC offset voltage is returned. If the IchOffsetEnable flag is set to FALSE, 0 will be returned. If the hardware doesn't support the DC offset, 0 will also be returned.

USAGE:

Result = Pstat.SetIchOffset(Level)

Level	REAL	Offset voltage level. Both current and voltage
-------	------	--



channel offsets are specified in terms of voltage. Current values can be converted to voltages by multiplying by the potentiostat effective I/E resistance (See Pstat.IEResistor).

Result	REAL	The active (after any change) offset voltage. This value should be checked since some configurations will always return 0. If the IchOffsetEnable flag is set to FALSE, 0 will be returned.
--------	------	---

## Pstat.IchOffsetEnable()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Returns whether an offset for the I channel is enabled or disabled.

**USAGE:**

```
Result = Pstat.IchOffsetEnable()
```

Result	BOOL	TRUE = Enabled FALSE = Disabled
--------	------	------------------------------------

## Pstat.SetIchOffsetEnable()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Enable or Disable post potentiostat offset correction. This allows the DC component of a measured voltage signal to be removed. Removal of this DC component allows multiplication of the signal by a post gain factor (1x, 10x, 100x) while keeping this post gain signal within the voltage limit of the analog to digital converter (ADC). The effective resistance of the I/E Resistor must be taken into account to determine the voltage seen by the ADC for a corresponding current level.

**USAGE:**

```
Result= Pstat.SetIchOffsetEnable(State)
```

State	BOOL	TRUE = Enable FALSE = Disable
Result	BOOL	Returns State as Set

## Pstat.IchRange()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Report the current channel range. The measured current is converted into a voltage using the I/E Converter. This function reports the code for the full scale A/D range in use.

**USAGE:**

RangeSet = Pstat.IchRange()

RangeSet	INDEX	Voltage for 30000 counts on A/D
	0	0.03 V range
	1	0.30 V range
	2	3.00 V range
	3	30.00 V Range ( PCI4 family ONLY! )

## Pstat.SetIchRange()

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Sets and reports the current channel range. The function Pstat.SetIchRange can used either with an absolute range ( INDEX ) or a measured voltage ( REAL ). Setting the IchRange using a REAL voltage is preferred. The measured current is converted into a voltage on the I channel using the I/E converter.

**USAGE:**

RangeSet = Pstat.SetIchRange(Range)

Range	REAL	Maximum V to measure, function will pick appropriate range ( Preferred ).
	INDEX	The code for the desired range ( Alternate usage ). See Table, above.
RangeSet	INDEX	The code for the new current range setting. See Table, above.

## Pstat.IEResistor()



*Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Calculate the effective I/E resistance. This resistance, when multiplied by the cell current, gives the voltage output of the I/E converter.

**USAGE:**

Conversion = Pstat.IEResistor(Range)

Range INDEX Current Range

Conversion REAL Effective Resistance (Ohms).

## Pstat.InitSignal()



*Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Initialize the signal which is currently being used by the potentiostat. which signal is to be used by the potentiostat. This causes the signal to be reset to the beginning, so any curves run after the call to InitSignal will start with the beginning of the applied signal. This call should be made to make sure the signal being used by the potentiostat is in a known state.

USAGE:

Pstat.InitSignal ()

## Pstat.LEDS



### *Related Topics*

TYPE: INSTANCE FUNCTION ( PCI4 FAMILY ONLY! )

PURPOSE: Read the current LEDS setting.

See Pstat.SetLEDS() for more information.

USAGE:

Setting = Pstat.LEDS()

Setting BITS The current state of the LEDS in the 4 least significant bits.

## Pstat.SetLEDS

TYPE: INSTANCE FUNCTION ( PCI4 FAMILY ONLY! )

PURPOSE: Set the on/off state of the LEDs on a PCI4 family potentiostat.

Individual LEDS can be set or all 4 LEDS can be controlled simultaneously. The LEDS are labeled LED0, LED1, LED2 and LED3.

There are two different ways to call this function. In the first syntax, BITS arguments are used to show the desired bit pattern and to show the range of bits to be changed. Remember that a BITS constant has the form 0x0XXXX with X representing any hexadecimal digit.

Each of the lowest 4 bits of the BITS argument corresponds to one of the LEDS. A bit will only be changed if the mask argument has a one in that bit's bit position. A mask argument of 0x0003 would allow changes in LED0 and LED1. With this mask value, LED2 and LED3 will not be changed regardless of the bit pattern argument.

In the second calling syntax 4 BOOL parameters list the state of each bit. This format may be easier for the average user to understand.

USAGE:

Pstat.SetLEDS(Setting, Mask)

Setting BITS New output code in the lowest 4 bits.

A one in this argument causes the LED to be turned on.

Mask BITS Identify the bits to be changed.

or

Pstat.SetLEDS(Bit0, Bit1, Bit2, Bit3)

BitN BOOL The bit setting.

TRUE Turn the LED on

FALSE Turn the LED off

NIL Don't change this LED

## Pstat.MeasureA()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Read the voltage on the potentiostat's Auxiliary A/D Input.

The full scale range of the Auxiliary A/D input is  $\pm 3$  volts. The bit resolution is 0.1 mV. You can expect about 1 bit rms noise in the reading. This noise will average out if you take multiple readings. If you need to read a higher voltage, a resistive voltage divider can be placed in front of the high impedance differential input.

**USAGE:**

Reading = Pstat.MeasureA()

Reading REAL The input voltage. Units are volts.

## Pstat.MeasureI()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Read the cell current. This function is used to get an immediate current reading without the complexity of running a data acquisition curve.

The current range is autoranged if necessary to get an accurate reading. Current interrupt IR compensation is not performed. All potentiostat settings (filters, control modes, etc.) except the current range are left alone as the reading is taken. The cell switch is not turned on if it is off.

**USAGE:**

Current = Pstat.MeasureI()

Current REAL The cell current.

## Pstat.MeasureV()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Read the cell voltage. This function is used to get an immediate voltage reading without the complexity of running a data acquisition curve.

The voltage measurement range is autoranged if necessary to get an accurate reading.

Current interrupt IR compensation is not performed. All potentiostat settings (filters, control modes, etc.) except the voltage measurement range are left alone as the reading is taken. The cell switch is not turned on if it is off.

**USAGE:**

Voltage = Pstat.MeasureV()

Voltage REAL The cell voltage.

## Pstat.ModelNo()



### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Returns the Pstat Model Number.

USAGE:

```
PstatNo = Pstat.ModelNo()
```

PstatNo INDEX Pstat model number

## Pstat.Open()



### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Acquire the use of a given potentiostat. If the potentiostat doesn't exist or is being used in another experiment, report an error and return the value FALSE.

USAGE:

```
Status = Pstat.Open()
```

Status BOOL TRUE if successful  
FALSE if unsuccessful

## Pstat.PosFeedEnable()



### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Return the current status of the enable state of the Positive Feedback DAC.

USAGE:

```
Enable = Pstat.PosFeedEnable()
```

Enable            BOOL    TRUE - Positive Feedback is  
                              enabled  
  
                              FALSE - Positive Feedback is  
                              disabled

## Pstat.SetPosFeedEnable()



### *Related Topics*

TYPE: INSTANCE FUNCTION

**PURPOSE:** Set then enable state of the Positive Feedback DAC.

Positive feedback is enabled when performing positive feedback IR compensation. A call should also be made to SetPosFeedResistance to set the uncompensated resistance that will be used by the feedback loop.

**USAGE:**

```
Pstat.SetPosFeedEnable(Enable)
Enable          BOOL    TRUE - Enable Positive Feedback
                FALSE - Disable Positive
                Feedback
```

## Pstat.PosFeedResistance()



*Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Return the value of the uncompensated resistance in ohms.

**USAGE:**

```
Resistance = Pstat.PosFeedResistance()
Resistance  REAL    The value of the uncompensated
                    resistance in ohms.
```

## Pstat.SetPosFeedResistance()



*Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Sets the uncompensated resistance value used during positive feedback.

Positive feedback is enabled when performing positive feedback IR compensation. The positive feedback mode should be enabled prior to making this call.

**USAGE:**

```
Result = Pstat.SetPosFeedResistance(Resistance)
Resistance  REAL    The value of the uncompensated
                    resistance to set in ohms.
Result      REAL    The value of the uncompensated
                    resistance as set.
```

## Pstat.PrintI()



*Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Print which potentiostat is being used. The output from Pstat.PrintI() looks like:

```
Tag <tab> PSTAT <tab> Label <tab> Potentiostat
```

where the third field, Label, is the string used to display the name of the potentiostat .

USAGE:

Pstat.Printl(Tag)

Tag STRING Optional. Used to print a different Tag than the one used in the call to .New

## Pstat.ScanLimitAC()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Report the maximum RMS signal the Pstat can deliver.

USAGE:

Result = Pstat.ScanLimitAC()

Result	REAL	Maximum RMS signal that can be applied.
--------	------	---

## Pstat.Section()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Returns the Pstat Section String from GAMRY5.INI file. This is the string that identifies each Pstat in the INI file. e.g. "PCI4300-01234"

USAGE:

PstatString = Pstat.Section()

PstatString STRING Pstat section string

## Pstat.SenseSpeed()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Return which I/E electrometer (Fast or Slow) is currently being used (PCI4 Femtostat Only).

USAGE:

State = Pstat.SenseSpeed()

State

SenseSlow Slow I/E and Electrometer (more accuracy)

SenseFast Fast I/E and Electrometer (more speed)

## Pstat.SetSenseSpeed()

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Sets which I/E electrometer (Fast or Slow) is to be used (PCI4 Femtostat Only).

**USAGE:**

Pstat.SetSenseSpeed(State)

State

SenseSlow    Slow I/E and Electrometer (more accuracy)

SenseFast    Fast I/E and Electrometer (more speed)

## Pstat.SenseSpeedMode()



*Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Returns the status of the Sense Speed auto-set mode. ( PCI4 Femtostat ONLY! ).

**USAGE:**

State = Pstat.SenseSpeedMode()

State

TRUE        Sense Speed automatically selected based on current range

FALSE       Sense Speed selected by Pstat.SetSenseSpeed() for all current ranges

## Pstat.SetSenseSpeedMode()

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Sets the Sense Speed auto-set mode. ( PCI4 Femtostat ONLY! ).

**USAGE:**

Pstat.SetSenseSpeed(State)

State

TRUE        Sense Speed automatically selected based on current range

FALSE       Sense Speed selected by Pstat.SetSenseSpeed() for all current ranges

**INSTRUMENTS:** PCI4, PC5

## Pstat.SerialNo()



**Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Returns the Pstat Serial Number.

**USAGE:**

Serial = Pstat.SerialNo()

Serial INDEX Pstat serial number

**INSTRUMENTS:** PCI4, PC5

**Pstat.SetAchRange()****Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Sets the range of the Auxiliary electrometer. Currently this function performs no valid use on the PCI4 and PC4 potentiostats, as each of these potentiostats has only one Ach Range.

**USAGE:**

RangeSet = Pstat.SetAchRange (Range)

RangeSet	INDEX	Range of Aux Channel as Set
Range	INDEX	Range of Aux Channel, currently only 0 is used.
	REAL	Maximum voltage to be measured, function will select appropriate range

**Pstat.SetIchRangeMode()****Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Enable or disable current channel measurement autoranging.

If autoranging is enabled (on), after each current channel measurement is made, the current channel measurement range is adjusted before the next point is recorded. The new range is calculated so that the measured voltage will optimally fill the A/D converter with little danger of overflow.

By default current channel autoranging is enabled.

**USAGE:**

Pstat.SetIchRangeMode(Mode)

Mode BOOL TRUE is autoranging to be enabled  
(default setting)  
FALSE if autoranging to be disabled.

## Pstat.SetIruptMode()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** SetIruptMode() controls current interrupt IR compensation. Using this function you can control:

- a) current interrupt timing
- b) voltage error calculation mode
- c) feedback mode

The current interrupt feature of the potentiostat uses a 3 sample algorithm to estimate the IR-free voltage. It first samples V at the given current level then it interrupts the current path and waits a time period, Tau. At that point it samples V again. It waits Tau seconds again and samples V for the third time. It then turns on the current.

The sample time, Tau, is increased as the current decreases. The Time parameter used in the call to SetIruptMode is used directly as Tau on the least sensitive current ranges. On the more sensitive current ranges, Tau is longer, but remains proportional to the Time parameter. Tau is constrained to be between 10 microseconds and 32,768 microseconds. Values outside this range are clipped to the nearest point within the range.

There are two different ways of calculating the IR voltage error. In calculation Mode 1, the three voltages, Vi, Voc1, and Voc2 are used to calculate an IR error voltage VIR via an extrapolation. In Mode 2, an average is used.

$$VIR = V_i - 2 * Voc1 + Voc2 \quad (\text{Calculation Mode 1})$$

$$VIR = V_i - 1/2 (Voc1 + Voc2) \quad (\text{Calculation Mode 2})$$

A third "calculation mode", Mode 0, is used to turn off the interrupt. In Mode 0, all of the other function parameters are ignored.

Once the IR error has been calculated, it can be used in several different ways, most of which involve feedback of the error signal.

In Galvanostat mode, there is never any IR compensation feedback. The feedback mode setting is therefore ignored in the Galvanostat mode. The measured error is used to correct potential measurements stored in the data curve.

In Potentiostat mode, there are 3 different feedback modes.

- a) No feedback.
- b) Normal feedback. The measured error voltage is added to the voltage applied for the next point.
- c) Control loop with fixed gain.

**USAGE:**

Pstat.SetIruptMode(FbMode, CalcMode, Timer, Eoc, Gain)

FbMode INDEX Feedback mode. Can use a predefined constant  
IruptOff = 0 (no feedback)  
IruptNorm = 1 (normal feedback)  
IruptClfg = 2 (control loop fixed gain)

CalcMode INDEX IR error calculation mode. Use predefined constants  
EuNone = 0 (no interrupt measurement)  
EuExtrap = 1 (extrapolation)  
EuAverage = 2 (average)

Timer REAL Nominal Tau value (units are seconds)  
This is the shorted time. Can be longer at low currents. Predefined constant:  
IruptTime = 50 e-6

Eoc REAL Open circuit E. Used only in control loop modes.

Gain REAL Cell gain. Used only in fixed gain control loop.

## Pstat.SetVchRangeMode()



**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Enable or disable voltage channel measurement autoranging.

If autoranging is enabled (on), after each voltage channel measurement is made, the voltage channel measurement range is adjusted before the next point is recorded. The new range is calculated so that the measured voltage will optimally fill the A/D converter with little danger of overflow.

By default voltage channel autoranging is enabled.

**USAGE:**

```
Pstat.SetVchRangeMode(Mode)
```

Mode BOOL TRUE is autoranging to be enabled (default setting)  
FALSE if autoranging to be disabled.

## Pstat.Sprint()



**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Print the potentiostat object in a string. The string will contain just the potentiostat number.

**USAGE:**

```
Result = Pstat.Sprint()
```

Result STRING Potentiostat board number.

## Pstat.Stability()



**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Report the potentiostat I/E Stability setting.

**USAGE:**

```
Switch = Pstat.Stability()
```

Switch INDEX The setting.

## Pstat.SetStability()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Set the I/E converter stability for potentiostat mode. There are four settings for stability, Fast (0), MediumFast (1), Medium or Normal (2), and Slow (3) with the most stable case being Slow. The Fast value corresponds to no extra I/E converter filtering. The Medium value corresponds to a small I/E filter which enhances stability. The Slow value corresponds to a large I/E filter which removes 50/60 Hz noise in the lower current ranges.

While it would take an advanced course in amplifier design to completely understand the stability setting, we can offer you a few guidelines:

1. Always leave the stability in Fast for galvanostatic operation.
2. If the potentiostat is showing high speed oscillation that depends on the current range in use, try increasing the Stability setting.
3. If your curves are noisy at low currents, try setting Stability to Slow.
4. If you are acquiring data faster than 0.2 seconds/point and see glitches when the current range changes, try decreasing the stability.

**USAGE:**

SetSwitch = Pstat.SetStability(Switch)

Switch INDEX

- 0 = No IE filtering (Fast)
- 1 = very small filter (Medium Fast)
- 2 = small filter (Medium, Normal)
- 3 = large filter (Slow)

SetSwitch INDEX The new setting.

Four constants have been set to make code reading easier. Located in the pstat.exp script, their names are:

<i>StabilityFast</i>	0
<i>StabilityMedFast</i>	1
<i>StabilityNorm</i>	2
<i>StabilitySlow</i>	3

## Pstat.TestAchRange()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Returns the optimum Aux Channel range given a voltage. Currently there is only one aux channel range, so this range (0) will always be returned.

**USAGE:**

Range = Pstat.AchRange (Voltage)

Voltage                      REAL                      Maximum voltage to be measured

Range	INDEX	Range of Aux Channel, currently only 0 is used.
-------	-------	---

### Pstat.TestAchRangeAC()



#### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Returns the optimum Aux Channel range given a voltage and a frequency. This function can return a less sensitive range based upon the frequency of acquisition, and the time constant of the range.

**USAGE:**

Range = Pstat.AchRange (Voltage, Frequency)

Voltage	REAL	Maximum voltage to be measured
---------	------	--------------------------------

Frequency	REAL	Frequency of measurement
-----------	------	--------------------------

Range	INDEX	Range of Aux Channel
-------	-------	----------------------

### Pstat.TestIchFilter()



#### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Report an index stating the appropriate IchFilter setting given a desired frequency. This function is automatically called when the SetIchFilter function is used. The potentiostat will return an index specifying the best filter setting which will give information on the frequency specified.

**USAGE:**

Setting = Pstat.TestIchFilter(Freq)

Freq	REAL	Frequency of interest in Hz
------	------	-----------------------------

Setting	INDEX	Best filter setting to use for the specified frequency. See Pstat.SetIchFilter.
---------	-------	---

### Pstat.TestIchRange()



#### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Finds the optimum voltage range of the current channel to maximize the resolution of the A/D converter.

**USAGE:**

Range = Pstat.TestIchRange(Voltage)

Voltage	REAL	Maximum anticipated voltage.
---------	------	------------------------------

Range	INDEX	Optimum voltage range for I channel
-------	-------	-------------------------------------

**Pstat.TestVchFilter()****Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Report an index stating the appropriate VchFilter setting given a desired frequency. This function is automatically called when the SetVchFilter function is used. The potentiostat will return an index specifying the best filter setting which will give information on the frequency specified.

**USAGE:**

Setting = Pstat.TestVchFilter(Freq)

Freq	REAL	Frequency of interest in Hz
Setting	INDEX	Best filter setting to use for the specified frequency. See Pstat.SetVchFilter.

**Pstat.TestVchRange()****Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Finds the optimum voltage range of the voltage channel to maximize the resolution of the A/D converter.

**USAGE:**

Range = Pstat.TestVchRange(Voltage)

Voltage REAL Maximum anticipated voltage.

Range INDEX Optimum voltage range for V channel

**Pstat.TestVchRangeAC()****Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Finds the optimum voltage range of the voltage channel to maximize the resolution of the A/D converter. Takes into account a higher than normal measurement frequency.

**USAGE:**

Range = Pstat.TestVchRangeAC(Voltage, Frequency)

Voltage REAL Maximum anticipated voltage.

Frequency REAL Measurement Frequency

Range INDEX Optimum voltage range for V channel

### Pstat.ThermoSelect()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Report the current ThermoSelect setting.

USAGE:

Setting = Pstat.SetThermoSelect()

Setting	INDEX	Type of value reported when measured
		0 = Voltage
		1 = Degrees Celcius

INSTRUMENTS: PC5

### Pstat.SetThermoSelect()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Pstat.SetThermoSelect specifies the type of value to be reported when the Auxiliary channel on the control board is set to measure the Thermocouple.

USAGE:

ThermoSelectSet = Pstat.SetThermoSelect (Type)

Type	INDEX	Type of value to report when measured
		0 = Voltage
		1 = Degrees Celcius

ThermoSelectSet	INDEX	The new type to be reported
-----------------	-------	-----------------------------

INSTRUMENTS: PC5

### Pstat.VchFilter()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Report current setting of Voltage Filter.

USAGE:

```
Filter = Pstat.VchFilter()
```

## Pstat.SetVchFilter()

*Related Topics*TYPE: INSTANCE FUNCTIONPURPOSE: Pstat.SetVchFilter() sets the filter on the voltage measuring channel. This filter has no effect on potentiostat stability.

If the parameter passed by SetVchFilter is a frequency in Hz, the potentiostat will then select the most appropriate filter setting able to pass information corresponding to this frequency.

NOTE: The No Filter setting (0) should not be used under normal circumstances. It is to be used exclusively for AC impedance measurements beyond 100 kHz.

USAGE:

```
FilterSet = Pstat.SetVchFilter(Freq)
```

Freq	REAL or INDEX	Frequency of interest in Hertz  Filter to select
		0 = No Filter
		0 = 200 kHz
		1 = 200 kHz
		2 = 1 kHz
		3 = 5 Hz
	or BOOL	FALSE = Ground. No Pass Filter setting.
FilterSet	INDEX	The new filter setting

## Pstat.VchRange()

*Related Topics*TYPE: INSTANCE FUNCTIONPURPOSE: Report the voltage channel range.USAGE:

```
RangeSet = Pstat.VchRange()
```

RangeSet	INDEX	Voltage for 30000 counts on A/D
	0	0.03 V range
	1	0.30 V range
	2	3.00 V range



## Pstat.SetVchRange()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Set voltage channel range. The function Pstat.SetVchRange can set either an absolute range or a measured voltage.

**USAGE:**

RangeSet = Pstat.SetVchRange(Range)

Range	REAL	Maximum V to measure, function will pick appropriate range ( Preferred ).
	INDEX	The code for the desired range ( Alternate usage ). See Table, above.
RangeSet	INDEX	The code for the new voltage range setting. See Table, above.

## Pstat.SetVoltage()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Provide a way to quickly and easily set the cell voltage in Potentiostat mode.

This function does not use or allow for current interrupt IR compensation.

The voltage will not be applied unless the cell switch is on.

Do not call this function with the hardware in Galvanostat mode.

**USAGE:**

Pstat.SetVoltage(Voltage)

Voltage REAL The voltage applied (working versus reference).

## Pstat.TestIchRangeAC()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Finds the optimum voltage range of the current channel to maximize the resolution of the A/D converter. Takes into account a higher than normal measurement frequency.

**USAGE:**

Range = Pstat.TestIChRangeAC(Voltage, Frequency)

Voltage REAL Maximum anticipated voltage.

Frequency REAL Measurement Frequency

Range INDEX Optimum voltage range for I channel

### Pstat.TestIERange()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Finds the optimum current range of the I/E converter for a given current.

USAGE:

Range = Pstat.TestIERange(Current)

Current                      REAL                      Maximum anticipated current.

If Abs(Current) is greater than I<sub>max</sub> for the selected potentiostat, the maximum possible range for that potentiostat will be returned.

Range                      INDEX                      Absolute current range (Full Scale Limit)

- 0 = 3 pA
- 1 = 30 pA
- 2 = 300 pA
- 3 = 3 nA
- 4 = 30 nA
- 5 = 300 nA
- 6 = 3 uA
- 7 = 30 uA
- 8 = 300 uA
- 9 = 3 mA
- 10 = 30 mA
- 11 = 300 mA
- 12 = 3 A
- 13 = 30 A
- 14 = 300 A
- 15 = 3 kA

### Pstat.TestIERangeAC()



*Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Find the optimum current range required to make a current measurement. This function takes into account the measurement frequency since a higher than normal current range is required at high frequencies. It also considers the maximum estimated voltage to avoid very large common mode voltages. The actual current range is not changed.

USAGE:

IRange = Pstat.TestIERangeAC(Iac, Eac, Idc, Edc, Frequency)

Iac REAL Maximum absolute AC current.

Eac REAL Maximum absolute AC voltage.

Idc REAL Maximum absolute DC current.

Edc REAL Maximum absolute DC voltage.

Frequency REAL Measurement frequency

IRange INDEX Resulting suggested current range.

### Pstat.VchOffset()



**Related Topics**

TYPE: INSTANCE FUNCTION

PURPOSE: Report the offset voltage of the voltage channel.

USAGE:

Result = Pstat.VchOffset ()		
Result	REAL	The offset voltage.

### Pstat.SetVchOffset()



**Related Topics**

TYPE: INSTANCE FUNCTION

PURPOSE: Set the DC offset voltage in the voltage channel. The actual offset circuitry may be implemented in the potentiostat or in the FRA. Not all hardware configurations have this feature. The actual DC offset voltage is returned. If the VchOffsetEnable flag is set to FALSE, 0 will be returned. If the hardware doesn't support the DC offset, 0 will also be returned.

USAGE:

Result = Pstat.SetVchOffset (Level)		
Level	REAL	Offset voltage level. Both current and voltage channel offsets are specified in terms of

		voltage. Current values can be converted to voltages by multiplying by the potentiostat effective I/E resistance (See <a href="#">Pstat.IEResistor</a> ).
Result	REAL	The active (after any change) offset voltage. This value should be checked since some configurations will always return 0. If the VchOffsetEnable flag is set to FALSE, 0 will be returned.

## Pstat.VchOffsetEnable()



**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Returns whether an offset for the V channel is enabled or disable.

**USAGE:**

Result = Pstat.VchOffsetEnable()		
Result	BOOL	TRUE = Enabled FALSE = Disabled

## Pstat.SetVchOffsetEnable()



**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Enable or Disable post potentiostat offset correction. This allows the DC component of a measured voltage signal to be removed. Removal of this DC component allows multiplication of the signal by a post gain factor (1x, 10x, 100x) while keeping this post gain signal within the +/- 3V limit of the analog to digital converter (ADC).

**USAGE:**

Result= Pstat.SetVchOffsetEnable(State)		
State	BOOL	TRUE = Enable FALSE = Disable
Result	BOOL	Returns State as Set

## Pstat.SetScanRange()



**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Set the Voltage range of the Scan DAC. When the signal is attached to the potentiostat, the Scan Range is automatically set. However, if you intend to set the Scan voltage manually via Pstat.SetScan(), a call to Pstat.SetScanRange() should be used beforehand.

**USAGE:**

Result = Pstat.SetScanRange(V)

or

Result = Pstat.SetScanRange(V1,V2)

V	REAL or INDEX	Scan from -V to +V Set Range using predefined constants <i>ScanRangeCoarse = 0</i> <i>ScanRangeMed = 1</i> <i>ScanRangeFine = 2 (Default State)</i>
V1, V2	2 REALS	Scan from V1 to V2
Result	INDEX	Range as set.

### Pstat.SetBias()



#### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Set the applied bias voltage. This signal is first summed with the Scan, DDS signal, and External signal. It is then sent to the potentiostat's signal input. In potentiostat mode this sum is applied to the cell. In galvanostat mode the applied current is given by:

$$I_{cell} = I_{range} * (V/3)$$

where V = Bias to be applied (Real Number)

*NOTE:* This function is usually superceded by the Signal Generator.

USAGE:

Result = Pstat.SetBias(Bias)

Bias REAL Bias to be applied (Volts)

Result REAL Returns Bias as set

### Pstat.SetScan()



#### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Set the scan voltage. This signal is first summed with the Bias, DDS signal, and External signal. It is then sent to the potentiostat's signal input. In potentiostat mode this sum is applied to the cell. In galvanostat mode the applied current is given by:

$$I_{cell} = I_{range} * (V/3)$$

where V = Bias to be applied (Real Number)

*NOTE:* This function is usually superceded by the Signal Generator.

USAGE:

Result = Pstat.SetScan(Bias)

Bias REAL Bias to be applied (Volts)

Result REAL Returns Bias as set

## Pstat.SetSignal()



*Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Specify which signal is to be used by the potentiostat. Multiple signals can be pre-created, and then used one at a time by making a call to SetSignal.

**USAGE:**

Pstat.SetSignal (Signal)

Signal	Object	Signal is a signal object created by a call to one of the signal creation class functions such as <a href="#">VCONST.New</a> . Different signals are available in different packages.
--------	--------	---

## PSTATSELECT



*Related Topics*

**TYPE:** CLASS

**PURPOSE:** This class is used to make selecting a potentiostat device independent. By using a device independent class for selecting a potentiostat, scripts do not need to know the family of potentiostat(s) in the computer. The PSTATSELECT class retrieves information from the GAMRY.INI file for the potentiostat(s) installed in the computer. A selector is usually passed to setup so the end user can select from the list of available potentiostats. Once the user has selected a potentiostat, the PSTATSELECT class returns a potentiostat of the appropriate family.

## PSTATSELECT.New



*Related Topics*

**TYPE:** CLASS FUNCTION

**PURPOSE:** Create and initialize a new PSTATSELECT object.

**USAGE:**

Pstatselect = PSTATSELECT.New (Tag, Prompt)		
Tag	STRING	Used by Setup Save/Restore
Prompt	STRING	Used by <a href="#">Setup()</a> to prompt operator for input
Pstatselect	PSTATSELECT	The PSTATSELECT object

## Pstatselect.CreatePstat



*Related Topics*

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Create a PSTAT object of the appropriate type for the potentiostat selected by the user during Setup(). In order for the user to be

able to select a potentiostat, a call to `Pstatselect.Selector()` needs to be passed to the [Setup\(\)](#) function.

#### USAGE:

<code>Pstat = Pstatselect.CreatePstat (Tag, Key)</code>		
Tag	STRING	The Tag associated with the Pstat object. Used by Setup Save/Restore
Key	STRING	The Key in the GAMRY.INI file which contains the information regarding the class of the selected potentiostat.
Pstat	PSTAT	The PSTAT object

### **Pstatselect.Selector**



#### *Related Topics*

#### TYPE: INSTANCE FUNCTION

**PURPOSE:** Create a selector object which contains a list of the potentiostats currently installed in the system. This function is usually used in conjunction with the [Setup\(\)](#) function to allow the user to select a potentiostat for use in an experiment. The Style information is used only when one or more of the potentiostats in the system is already in use. If a potentiostat is already in use, its name will have the selected Style.

#### USAGE:

<code>Selector = Pstatselect.Selector (Style)</code>		
Style	INDEX	The style to use when a potentiostat is already in use.
		0 = Normal (No special style)
		1 = Asterisk (* appears after label)
		2 = Disabled (label grayed out)
		or use the predefined constants
		<i>SELECTOR_NORMAL</i>
		<i>SELECTOR_ASTERISK</i>
		<i>SELECTOR_DISABLED</i>
Selector	SELECTOR	The selector object

### **Pstatselect.FraCurveClass**



#### *Related Topics*

#### TYPE: INSTANCE FUNCTION

**PURPOSE:** Return the FRACURVE class for the potentiostat selected by the user during `Setup()`. In order for the user to be able to select a potentiostat, a call to `Pstatselect.Selector()` needs to be passed to the [Setup\(\)](#) function. This function is used mainly for impedance scripts which will use either an external or built-in FRA. This function is used to allow for device independent scripts.

#### USAGE:

`FRACURVE = Pstatselect.FraCurveClass (Key)`

Key	STRING	The Key in the GAMRY.INI file which contains the information regarding the FRACURVE class of the selected potentiostat. (Usually "FraCurveClass")
FRACURVE	CLASS	The FRACURVE class of the selected potentiostat.

### Pstatselect.PstatClass



#### *Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Return the PSTAT class for the potentiostat selected by the user during Setup(). In order for the user to be able to select a potentiostat, a call to Pstatselect.Selector() needs to be passed to the [Setup\(\)](#) function. This function is used to allow for device independent scripts.

USAGE:

PSTAT = Pstatselect.PstatClass (Key)		
Key	STRING	The Key in the GAMRY.INI file which contains the information regarding the PSTAT class of the selected potentiostat. (Usually "PstatClass")
PSTAT	CLASS	The PSTAT class of the selected potentiostat.

### QUANT



#### *Related Topics*

TYPE: CLASS

PURPOSE: The QUANT class allows a user to edit, save and restore REAL values in Setup(). Remember, REAL's, by themselves, cannot be given as parameters to Setup() since Explain uses Call by Value to pass parameters. A REAL must therefore be encapsulated in a QUANT to be used in Setup.

### QUANT.New()



#### *Related Topics*

TYPE: CLASS FUNCTION

PURPOSE: Create and initialize a new Quant object.

USAGE:

```
Quant = QUANT.New(Tag, Quantity, Prompt)
```

Tag STRING Used by Setup() Save/Restore,  
Tag.Printl()



Quantity REAL Initial value for Quant

Prompt STRING Used by Setup() to prompt operator for input

Quant QUANT The initialized object

## Quant.Printl()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Print a Quant to the current Output file. The Quant will be printed in the format:

Tag <tab> QUANT <tab> Quantity <tab> Prompt

If the Tag is empty Quant.Printl() will print only the Quantity.

For example, if the Quant was created with:

```
Quant = QUANT.New("VOLTAGE", 1.2, "Voltage")
```

then Quant.Printl() would give

```
VOLTAGE <tab> QUANT <tab> 1.200 <tab> Voltage
```

USAGE:

```
Quant.Printl(Tag)
```

Tag STRING Optional. Used to print a different Tag than the one used in the call to .New

## Quant.Value()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Copy the current value of a Quant into a REAL.

USAGE:

```
Value = Quant.Value()
```

Value REAL Current value of the Quant

## Quant.SetValue()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Copy a REAL into a QUANT.

USAGE:

Quant.SetValue(Quantity)

Quantity REAL New value of the Quant

**Quant.Sprint()***Related Topics*

TYPE: INSTANCE FUNCTION

PURPOSE: Print the Quant as a formatted string.

USAGE:

Result = Quant.Sprint(Format)

Format STRING Optional. Format string used in C-language sprintf statement. If missing, %G format is used.

Result STRING Resulting character string.

**QUANT.Dialog***Related Topics*

TYPE: LICENSE

PURPOSE: Allow Quant to be edited, saved, and restored in Setup(). The Quant will be edited using the fields:

Prompt [Quantity ]

When the operator enters a value, Setup will attempt to turn it into a valid REAL and replace the Value in Quant with it. For example, if an operator puts in "1", Setup() will change it to "1.0". If Setup() cannot translate a string into a REAL, it will default to the value 0.0.

**Query()***Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Ask the operator to answer a multiple choice question. A dialog box with the choices is displayed, and the script pauses until the operator makes a choice.

The number of choices in a query dialog box is limited by the area on the screen available for the dialog box. More than 10 choices are not recommended.

USAGE:

Answer = Query(Question, Choice1, Choice2, ..., ChoiceN)

Answer INDEX A code telling which choice was selected. The answer number is zero based. The first choice therefore returns zero not one.

Question STRING The overall question asked.

ChoiceN STRING The label for the Nth choice.

## Rand()



### *Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Calculate a random number from 0 to RAND\_MAX (32767). Several options can occur, however. If no number is passed, a random number from 0 to RAND\_MAX is returned. If a negative number is passed, RAND\_MAX is returned. If a positive number is passed, the random number generator is re-seeded.

USAGE:

Result = Rand(Number)

Number INDEX NIL or NEGATIVE or POSITIVE

Result INDEX Random number if Number is NIL  
 RAND\_MAX if Number is Negative  
 NIL if Number is Positive

## Real()



### *Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Convert INDEX, BITS, BOOL, or STRING to a REAL. Anything other than these legal arguments is converted to 0.000.

REAL() acting on string that does not contain a number returns an indeterminate value.

USAGE:

NewReal = Real(Quantity)

NewReal REAL The result.

Quantity INDEX, BITS, BOOL, The quantity to be  
 STRING floated

## Round()



### *Related Topics*

TYPE: REGULAR FUNCTION

**PURPOSE:** Round a REAL value to a known number of decimal places.

Round (3.0149, 2) --> 3.01 (Rounded to 2 places)

Round (10.51, 0) --> 11 (Round to 0 places)

Round (-1.06,1) --> -1.1 (Round to 1 place)

**USAGE:**

Rounded = Round(Number, Places)

Rounded REAL The result.

Number REAL The quantity to be rounded.

Places INDEX The number of decimal places desired.

## RtoD()



### *Related Topics*

**TYPE:** REGULAR FUNCTION

**PURPOSE:** Convert an angle in Radians to Degrees.

**USAGE:**

Degrees= RtoD(Radians)

Radians REAL The angle in Radians.

Degrees REAL The angle in Degrees

## RunnerStatus()



### *Related Topics*

**TYPE:** REGULAR FUNCTION

**PURPOSE:** Write a short string in the runner status region of the Runner window. This region is on the status bar to the right of the notify region. This is usually written to by the Explain DLL and explain programmers should use Notify or Notify2 for their notifications.

**USAGE:**

RunnerStatus(Text)

Text STRING

# SELECTOR



## Related Topics

[TYPE:](#) CLASS

[PURPOSE:](#) The SELECTOR class allows a user to select a value from a list of values in Setup(). The value can be saved in and restored from the .SET file. The selector object is presented to the user as a set of up to 8 radio buttons, only one of which will be selected. See the Selector instance functions in subsequent sections.

# SELECTOR.New()



## Related Topics

[TYPE:](#) CLASS FUNCTION

[PURPOSE:](#) Create a Selector object.

[USAGE:](#)

Selector = SELECTOR.New(Tag, List, Value, Prompt)		
Tag	STRING	Used in Setup for Save/Restore, Printing. Tag is a name associated with the object, itself, and not with the variable.
List	VECTOR	Vector of strings, one for each possible selection. The strings should be short to be displayed in a limited space. The length of List governs the number of positions in the selector.
Value	INDEX	Which item is selected. Value is 0 based, i.e. when the first item is selected the Value is 0
Prompt	STRING	Used is Setup() for data entry
Selector	SELECTOR	Newly created SELECTOR object.

# Selector.Value()



## Related Topics

[TYPE:](#) INSTANCE FUNCTION

[PURPOSE:](#) Copy the current value of Selector into an INDEX.

[USAGE:](#)

Result = Selector.Value()		
Result	INDEX	Currently selected item in the list. First item has Value=0.

# Selector.Label()



## Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Retrieve the selected label from a Selector.

USAGE:

```
Result = Selector.Label()
```

Result STRING Name of the currently selected item in Selector.

## Selector.SetValue()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Set the value of a selector

USAGE:

```
Selector.SetValue(Value)
```

Value INDEX New value to be set.

## Selector.Printl()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Print the selector to the output file. Selector will be printed in the format:

```
Tag <tab> SELECTOR <tab> Value <tab> Prompt
```

USAGE:

```
Selector.Printl(Tag)
```

Tag STRING Optional. Used to print a different Tag than the one set in the call to SELECTOR.New

## Selector.Sprint()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Print a selector value to a string.

USAGE:

```
Result = Selector.Sprint()
```

Result STRING Index number of the current selection.

## Selector.SetStyle()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Format the style of a selector. This function allows the user to set certain selector items to be asterisked or grayed out. The following table shows the use of bits to format positions within the selector. For selectors with more than 4 positions, a two or three digit hexadecimal number (Bits) will be required.

#### BINARY BITS RESULT

0000 0x0 All positions enabled  
 0001 0x1 Position 1 disabled  
 0010 0x2 Position 2 disabled  
 0011 0x3 Positions 1 and 2 disabled  
 0100 0x4 Position 3 disabled  
 0101 0x5 Positions 1 and 3 disabled  
 0110 0x6 Positions 2 and 3 disabled  
 0111 0x7 Positions 1, 2, and 3 disabled  
 1000 0x8 Position 4 disabled  
 1001 0x9 Positions 1 and 4 disabled  
 1010 0xA Positions 2 and 4 disabled  
 1011 0xB Positions 1, 2, and 4 disabled  
 1100 0xC Positions 3 and 4 disabled  
 1101 0xD Positions 1, 3, and 4 disabled  
 1110 0xE Positions 2, 3, and 4 disabled  
 1111 0xF Positions 1, 2, 3, and 4 disabled

**USAGE:**

Selector.SetStyle(Pattern, Style)

Pattern BITS Pattern of the selector to be used.

Style INDEX Style of Disabled position

0 = Normal - Not disabled

1 = Asterisk - Asterisk next to name

2 = Disabled - Grayed out, cannot be selected.

Or use the following constants:

*SELECTOR\_NORMAL*

*SELECTOR\_ASTERISK*

*SELECTOR\_DISABLED*

## SELECTOR.Dialog



### Related Topics

**TYPE:** LICENSE

**PURPOSE:** Allow Selector to be edited, saved, and restored in Setup(). The Selector will be edited using the format:

```
Prompt  [X ] Label0  [ ] Label1  [ ] Label2  [ ] Label3
         [ ] Label4  [ ] Label5  [ ] Label6  [ ] Label7
```

## SetConfig()



### Related Topics

**TYPE:** REGULAR FUNCTION

**PURPOSE:** Write a value to an INI file. After the write there will be an entry in the INI file of the form:

```
[Section]
Key=Text
```

**USAGE:**

```
SetConfig(File,Section,Key,Text)
```

File	STRING	What file to write to. If the file doesn't have a path, the system looks for it in the same directory that contains the Gamry5.ini file. (Typically "C:\Documents and Settings\All Users\Application Data\Gamry Instruments\Framework")
Section	STRING	Section name within the file.
Key	STRING	Key name
Text	STRING	Value to write.
<i>or</i>	NIL	Delete the Key from the INI file.

## Setup()



### Related Topics

**TYPE:** REGULAR FUNCTION

**PURPOSE:** Setup() displays a dialog box where the user can modify, save, or recall script parameters.

The dialog box has a title which is specified by the function call. It may have a variable number of parameters. When Setup() is executed, a dialog box is opened.

All items in a Setup dialog box must be objects with a Dialog license. An License Error will be reported if you give Setup an argument that does not have this license. Simple data types (STRING, REAL, etc.) are invalid arguments. Refer to the individual class descriptions for Dialog license details.

Each object is displayed as a prompt and one or more fields which the user may modify. Each object in Setup() determines how the fields are displayed based on its dialog license. The object specifies the prompt to be used to identify itself.

The prompt can have an accelerator key. The accelerator key is shown as an underlined character in the prompt. When the accelerator key is typed simultaneously with the ALT key, the dialog box focus switches immediately to the object associated with that accelerator. Accelerator keys are identified by an ampersand (&) in the prompt for an object. For example, the sample period object can be created by the following function:

```
Sample = QUANT.New("SAMPLETIME", 0.5, "Sa&mple Period (s)")
```

The prompt contains an ampersand before the m in the prompt "Sa&mple Period (s)". When this prompt is shown in a Setup dialog box the m will be underlined. More importantly, the Setup() function automatically sets up ALT + M as the accelerator for this object.

Avoid use of the following characters as accelerators: R, O, S, C, D, N. These accelerators are already used in Setup() as accelerators for pushbuttons. All other letters can be used. If you run out of letters, one can be repeated. In this case, the accelerator cycles among all objects using this accelerator key.

When the **Save** key is hit, the objects in the Setup dialog are all saved to a Parameter Set stored in a Parameter File. The TAG field of each object is used as a parameter identifier in the Parameter File.

The return value from Setup depends on the key the user selects to exit the dialog box. Setup returns TRUE if the user leaves via **OK** and returns FALSE if the user leaves via **CANCEL**.

**USAGE:**

```
Status = Setup(Title, Item1, Item2, Item3, ...)
```

Title	STRING	This Setup dialog box title.
ItemN	Object	Object of any class having a Dialog License.
Status	BOOL	TRUE if user keys <b>OK</b> button to end Setup()



FALSE if user keys CANCEL button to end Setup().

## SetupRestore()



### Related Topics

**TYPE:** REGULAR FUNCTION

**PURPOSE:** SetupRestore() recalls a selected group of experimental parameters that have previously been stored in a disk file. It is equivalent to a Setup() function in which the user chooses to restore an experimental parameter set. The major differences are in SetupSave() where no dialog box is displayed and the user need not take any action to recall the setup parameters.

**NOTE:** In the Gamry Framework a file used to store experimental parameters can contain more than one set of parameters. Each set of parameters is stored in a separate section of the file. The sections are identified by a SetupName. In the ASCII Setup file, the SetupName that precedes each section is enclosed in square brackets (e.g. [Test 5] ). The value of each object is in a string of the form:

Tag = Data

As examples, these lines were taken from a saved Setup() file:

```
[POLRES]
PSTAT=0
TITLE=Polarization Resistance
OUTPUT=My data file.DTA
NOTES=1
NOTES_LINE_0=dummy cell
VINIT=-0.02, T
VFINAL=0.02, T
SCANRATE=0.2
```

An object must have a dialog license to be used in SetupRestore(). A License Error dialog box will be displayed if you attempt to SetupRestore() an invalid variable or object.

No error indication occurs if you try to restore an object and its Tag is not found in the SetupName section of the Setup file. The old setting for the object is not changed.

### USAGE:

```
Status = SetupRestore(FileName, SetName, Item1, Item2, Item3, ....)
```

FileName	STRING	Pathname to the Setup file.
SetName	STRING	Name of the section in the Setup file.
ItemN	Object	Object of any class having a Dialog License.
Status	BOOL	TRUE if Setup file opened and field name FALSE if file cannot be opened or field name invalid.

## SetupSave()




### Related Topics

**TYPE:** REGULAR FUNCTION

**PURPOSE:** SetupSave() stores a selected group of experimental parameters in a disk file. It is equivalent to a Setup() function in which the user chooses to save the current Setup() settings. The major differences are that in SetupSave() no dialog box is displayed and the user need not take any action to store the setup parameters.

In the Gamry Framework a file used to store experimental parameters can contain more than one set of parameters. Each set of parameters is stored in a separate section of the file. The sections are identified by a SetupName. In the ASCII Setup file, the SetupName that precedes each section is enclosed in square brackets (e.g. [Test 5] ). The value of each object stored in a section of the file is preceded by the object's Tag.

	An object must have a dialog license to be used in SetupSave(). An error dialog box will be displayed if you attempt to SetupSave () an invalid variable or object.
---	--

The Setup file may already have items stored in the SetupName section. If an item already stored in the file does not have the same Tag as an object being saved, that item is not changed. Old items with the same Tag as a new item are overwritten.

**USAGE:**

```
Status = SetupSave(FileName, SetupName, Item1, Item2, Item3, ....)
    FileName    STRING    Pathname to the Setup file.
    SetupName   STRING    Name of the section in the Setup file.
    ItemN       Object    Object of any class having a Dialog License.
    Status      BOOL      TRUE if Setup file opened and field name valid
                   FALSE if file cannot be opened or field name invalid.
```

## Signal.SetAcquisitionControl



**TYPE:** INSTANCE

**PURPOSE:** To allow the user to specify a specific data acquisition parameters. This instance function is available for all signals.

**USAGE:**

Signal.SetAcquisitionControl (AcqMode, AcqBasePeriod, SampleRate, DutyCycle)		
AcqMode	INDEX	Type of acquisition mode. 0 = Fast Mode  1 = DSP Mode
	or	
	NIL	Use the default AcqMode. This mode is set based on the output point sample period. Below 100.0 μs the system uses Fast Mode. At or above 100.0 μs, the system use DSP Mode.
AcqBasePeriod	REAL	The time between each sample point. This number should not be less than the default of 16.666 μs.
	or	
	NIL	Use the default AcqBasePeriod. For ACQ_MODE_FAST, this is not used. For ACQ_MODE_DSP, the default is 16.666 μs.
SampleRate	REAL	The output point rate, in seconds. Output points are the points displayed to the end user and saved to the curve. They are made up of 1 or more sample points.
	or	

	NIL	Use the default sample rate specified in the SIGNAL.New call.
DutyCycle	REAL	A number between from 0.0 to 1.0 specifying the duty cycle that is used in sampling. 0.0 will always give 1 sample, while 1.0 will sample as quickly as possible.
	or	
	INDEX	0 = 1 sample only (duty cycle of 0.0) 1 = Use the default duty cycle of 0.20 2 = Use a duty cycle of 1.0
	or	
	NIL	Use the default duty cycle of 0.20



Note: For a PCI4 family Potentiostat the parameters are ignored. Only the Fast Mode is available.

## Sin()



### Related Topics

TYPE: REGULAR FUNCTION

PURPOSE: Calculate the sine of an angle. The angle must be in radians.

USAGE:

Result = Sin(Angle)

Angle REAL The angle of which to calculate the Sine.

Result REAL The Sine of Angle.

## Sinh()



### Related Topics

TYPE: REGULAR FUNCTION

PURPOSE: Calculate the hyperbolic sine of a number. The definition of Sinh is:

$$\text{Sinh}(x) = [e^x - e^{-x}] / 2$$

USAGE:

Result = Sinh(Number)

Number REAL The number of which to calculate the hyperbolic sine.

Result REAL The hyperbolic sine of Number.

# Sleep()



## Related Topics

TYPE: REGULAR FUNCTION

PURPOSE: Pause the execution of this script until a specific time or until the user hits the **SKIP** button. Times used in Sleep() are in system time expressed as seconds since 00:00:00 1/1/70. See the description of the Time() function for useful information about system times.

This function will return control to Windows so that it may continue to process other windows and applications. If you have experiments running in other windows, they will continue to run normally.

A time delay for a known time interval can be implemented in 3 steps:

- 1) Call Time() to get the current time as an INDEX.
- 2) Add the desired delay in seconds.
- 3) Use this value in a call to Sleep().

This can be done in one line. Suppose we want to delay for 15 seconds:

```
Sleep( 15 + Time() )
```

USAGE:

```
Result = Sleep(Time)
```

Time	INDEX	Sleep until this time  (seconds since 1/1/70)
Result	INDEX	The number of seconds remaining when Sleep returned.  Result will be zero if Sleep() returned because Time was reached.  If the user pressed the <b>Skip</b> button, Result will be the number of seconds remaining until Time.

# Sprint()



## Related Topics

TYPE: REGULAR FUNCTION

PURPOSE: Print one or more values to a STRING variable. Similar to the Print() function, except that output is not directed to the OUTPUT device or file.

One use for this function is the automatic generation of filenames or labels consisting of a fixed name and a repetition counter e.g. EXPT1, EXPT2, EXPT3...

USAGE:

```
String = Sprint(Value, ...)
```

String STRING The output string.

Value REAL, INDEX, STRING, The value(s) to  
BOOL, BITS be printed.

See the description of the Print() command for detail on output formats.

## Sqrt()



### *Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Take to square root of a number.

USAGE:

Result = Sqrt(Number)

Number REAL Number to be square-rooted. If  
Number < 0, absolute value is used.

Result REAL Result of the calculation.

## STATIC



### *Related Topics*

TYPE: CLASS

PURPOSE: A class for a simple text object which can be placed in a Setup box. This object is not user editable.

## STATIC.New()



### *Related Topics*

TYPE: CLASS FUNCTION

PURPOSE: Create a Static object.

USAGE:

Static = STATIC.New("Text")

Text TEXT Any text. "Text" can be replaced with a  
string object.

Static STATIC

## STATIC.Dialog



### Related Topics

[TYPE:](#) LICENSE

[PURPOSE:](#) Allow Static to be displayed in Setup().

## StatsOne()



### Related Topics

[TYPE:](#) Regular Function

[PURPOSE:](#) Calculate statistical information on a set of data. The data set can include all of the data in a DATACOL object, or a subset of the data in a DATACOL object. The calculated statistics are returned as members of a Vector.

[USAGE:](#)

Result = StatsOne (DataCol, FirstPoint, LastPoint)

DataCol	DATACOL	The column of data upon which to calculate the statistics
FirstPoint	INDEX	The first point to include in the data set (Optional)
LastPoint	INDEX	The last point to include in the data set (Optional)
Result	VECTOR	<p>[ 0] N Sum(1)</p> <p>[ 1] Mean Sum(X)/N</p> <p>[ 2] StdVar Sum(X-Xm)^2/N-1</p> <p>[ 3] StdDev sqrt(StdVar)</p> <p>[ 4] 1st moment Sum(abs(X-Xm))/N</p> <p>[ 5] 2nd moment Sum((X-Xm)^2)/N</p> <p>[ 6] 3rd moment Sum((X-Xm)^3)/N</p> <p>[ 7] 4th moment Sum((X-Xm)^4)/N</p> <p>[ 8] PopVar Sum((X-Xm)^2)/N</p> <p>[ 9] PopDev sqrt(PopVariance)</p> <p>[10] RMS sqrt(Sum(X^2)/N)</p>

## StatsTwo()



### Related Topics

[TYPE:](#) Regular Function

[PURPOSE:](#) Calculate statistical information between two sets of data. The data set can include all of the data in two DATACOL

objects, or a subset of the data in the two DATACOL objects. The two DATACOL objects must be of the same length. The calculated statistics are returned as members of a Vector.

USAGE:

Result = StatsTwo (DataColX, DataColY, FirstPoint, LastPoint)

DataColX	DATACOL	The first column of data
DataColY	DATACOL	The second column of data
FirstPoint	INDEX	The first point to include in the data set (Optional)
LastPoint	INDEX	The last point to include in the data set (Optional)
Result	VECTOR	[ 0 ] N Sum(1)
		[ 1 ] XMean Sum(X) /N
		[ 2 ] YMean Sum(Y) /N
		[ 3 ] XVarp Sum(X-Xm) ^2 / N
		[ 4 ] YVarp Sum(Y-Ym) ^2 / N
		[ 5 ] Covar Sum(X-Xm) * (Y-Ym) /N
		[ 6 ] XSig Sqrt(XVar)
		[ 7 ] YSig Sqrt(YVar)
		[ 8 ] Correl Covar/(XSig*YSig) (Pearson's R)
		[ 9 ] XSlope dY/dX = Covar/Xvarp
		[10] XYint X@Y=0
		[11] YSlope dX/dY = Covar/Yvarp
		[12] YXint Y@X=0

## Stdout()



*Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Stdout sends output to the STDOUT window. This is a useful function when debugging a new Explain Script. Numerical values, text values, or any combination can be sent as output.

USAGE:

Stdout(variable)  
or  
Stdout("text")

## StdoutActivate()



*Related Topics*

[TYPE:](#) REGULAR FUNCTION

[PURPOSE:](#) StdoutActivate brings the STDOUT window to the foreground when running an Explain script. This is a useful function if text descriptors or numerical values are being displayed on the STDOUT window.

[USAGE:](#)

StdoutActivate()

## StrCmp()



*Related Topics*

[TYPE:](#) REGULAR FUNCTION

[PURPOSE:](#) Compares two strings character by character and returns a value indicating their relationship.

[USAGE:](#)

Result = StrCmp(String1, String2)

String1	STRING	First String for Comparison
String2	STRING	Second String for Comparison
Result	INDEX	-1 = String1 less than String2 0 = String1 identical to String2 1 = String1 greater than String2

## StrLwr()



*Related Topics*

[TYPE:](#) REGULAR FUNCTION

[PURPOSE:](#) Converts any uppercase letters in a string to lowercase.

[USAGE:](#)

AllLower = StrLwr(String)

String STRING String for conversion.

AllLower STRING String converted to all lowercase.

## StrUpr()



*Related Topics*

[TYPE:](#) REGULAR FUNCTION



**PURPOSE:** Converts any lowercase letters in a string to uppercase.

**USAGE:**

```
AllUpper = StrUpr(String)
```

String STRING String for conversion.

AllUpper STRING String converted to all uppercase.

## StrLen()



*Related Topics*

**TYPE:** REGULAR FUNCTION

**PURPOSE:** Returns the number of characters in a string, not including the terminating null character.

**USAGE:**

```
Length = StrLen(String)
```

String STRING String to determine length.

Length INDEX Number of characters in String.

## StrSet()



*Related Topics*

**TYPE:** REGULAR FUNCTION

**PURPOSE:** Sets a character in a string to a new ascii character specified by an index. Position is zero based.

e.g. String = "ABC"

```
StrSet(String, 2, 69)
```

String would now = "ABE"

**USAGE:**

```
StrSet(String, Position, AsciiNumb)
```

String	STRING	String to determine length.
Position	INDEX	Position within String to replace
AsciiNumb	INDEX	Character to overwrite the char in Position.

## StrGet()

**Related Topics**

[TYPE:](#) REGULAR FUNCTION

[PURPOSE:](#) Returns the Ascii number of a character in a string. Position within string is zero based.

[USAGE:](#)

AsciiNumb = StrGet(String, Position)

String	STRING	String containing character to determine
Position	INDEX	Position within string to get character
AsciiNumb	INDEX	Ascii number of character in Position

## Suspend()

**Related Topics**

[TYPE:](#) REGULAR FUNCTION

[PURPOSE:](#) Allows user to either suspend or continue execution. This function makes use of a query box which asks the user a question and gives the option to either Abort or Continue. The function takes one parameter which is either a string or text. This parameter is the message displayed in the query box.

[USAGE:](#)

Suspend(Message)

Message	TEXT/STRING	Message to display while waiting for user input
---------	-------------	---

## Tan()

**Related Topics**

[TYPE:](#) REGULAR FUNCTION

[PURPOSE:](#) Calculate the tangent of an angle. The angle must be in radians.

[USAGE:](#)

Result = Tan(Angle)

Angle REAL The angle of which to calculate the tangent.

Result REAL The tangent of Angle.

## Tanh()

**Related Topics**

TYPE: REGULAR FUNCTION

PURPOSE: Calculate the hyperbolic tangent of a number. The definition of Tanh is:

$$\text{Tanh}(x) = \text{Sinh}(x) / \text{Cosh}(x)$$

USAGE:

Result = Tanh(Number)

Number REAL The number of which to calculate the hyperbolic tangent.

Result REAL The hyperbolic tangent of Number.

## Time()



### *Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Calculate the number of seconds elapsed since 00:00:00 1/1/70 and the time specified by the parameter. This time convention is compatible with the Microsoft C time functions. If no parameters are specified, use the current system time and date. You can also substitute NIL for any parameter in the second and third form to use the current system values for that parameter.

USAGE:

Now = Time() if no parameters

or

Then = Time(DateStr, TimeStr) if 2 parameters

or

Then = Time(Mon, Day, Yr, Hr, Min, Sec) if 6 parameters

DateStr STRING of the form "MON/DAY/YR" or NIL  
year reported as number of years since 1900.

TmeStr STRING of the form "HR:MIN:SEC" or NIL

Mon INDEX (where 1 = January) or NIL

Day INDEX or NIL

Year INDEX (where 0 = 1900) or NIL

Hour INDEX or NIL

Min INDEX or NIL

Sec INDEX or NIL

Then INDEX number of seconds since 1/1/70 0:0:00

## TimeStamp()



TYPE: REGULAR FUNCTION

PURPOSE: Create a string of the form "Hr:Min:Sec". If no parameter is given, use the current system time. Otherwise, convert the parameter into a date. The parameter is derived from function Time().

You can print the current time and date to the output file by using:

```
Printl(TimeStamp(), " ", DateStamp())
```

USAGE:

```
Result = TimeStamp()
```

or

```
Result = TimeStamp(Marker)
```

Marker INDEX Elapsed seconds from 0:0:0, 1/1/70

Result STRING Time in the form Hr:Min:Sec

## TOGGLE



TYPE: CLASS

PURPOSE: Provide a simple ON/OFF switch that an operator may use to set a BOOL value in Setup. BOOL's cannot be edited directly in Setup (see discussion under class QUANT).

## TOGGLE.New()



TYPE: CLASS FUNCTION

PURPOSE: Create a Toggle object.

USAGE:

Toggle = TOGGLE.New(Tag, Switch, Prompt)

Tag	STRING	Used in Setup() for Save/Restore
Switch	BOOL	Initial value
Prompt	STRING	Used in Setup() for data entry
Toggle	TOGGLE	

## Toggle.Printl()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Print a Toggle to the current Output file. The Toggle will be printed in the format:

Tag <tab> TOGGLE <tab> T <tab> Prompt

or

Tag <tab> TOGGLE <tab> F <tab> Prompt

USAGE:

Toggle.Printl(Tag)

Tag	STRING	Optional. Used to print a different Tag than the one used in the call to Toggle.New
-----	--------	---

## Toggle.Value()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Copy the current value of a Toggle into a BOOL.

USAGE:

Switch = Toggle.Value()

Switch	BOOL	The current value of the Toggle
--------	------	---------------------------------

## Toggle.SetValue()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Copy a BOOL into an existing Toggle.

USAGE:

```
Toggle.SetValue(Switch)
```

Switch    BOOL    The new value of the Toggle

## TOGGLE.Dialog



### Related Topics

TYPE: LICENSE

PURPOSE: Allow Toggle to be edited, saved, and restored in Setup(). The Toggle will be edited using the fields:

Prompt [ ] Off or

Prompt [X] On

When the operator keys the button, Setup will change the current state of the Value and modify the display to reflect its state.

## Toggle.Sprint()



### Related Topics

TYPE: INSTANCE FUNCTION

PURPOSE: Print the toggle value out to a string.

USAGE:

```
Result = Toggle.Sprint()
```

Switch	BOOL	The current value of the Toggle
Result	STRING	Either T or F depending on state of Toggle.

## TWOPARAM



### Related Topics

TYPE: CLASS

PURPOSE: The TWOPARAM class allows the creation of a complex object consisting of one BOOL and two REALs bundled into one object. The advantage of bundling is that a Twoparam uses only one line in SETUP(). Both Conditioning and Initial delay can be SETUP using a Twoparam object.

## TWOPARAM.New()



### Related Topics

**TYPE:** CLASS FUNCTION

**PURPOSE:** Create and initialize a new Twoparam object.

**USAGE:**

```
Twoparam = TWOPARAM.New(Tag, Bval, Val1, Val2, Prompt, Prompt1, Prompt2)
    Tag          STRING      Used by Setup() Save/Restore, Tag.Printl().
    Bval         BOOL       Initial value for BOOL within Twoparam.
    Val1         REAL       Initial value for first REAL.
    Val2         REAL       Initial value for second REAL.
    Prompt       STRING     Prompt string for whole object.
    Prompt1     STRING     Prompt string for first REAL
    Prompt2     STRING     Prompt string for second REAL
    Twoparam    TWOPARAM   The initialized object.
```

## Twoparam.Printl()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Print a Twoparam to the current Output file. The Twoparam will be printed in the format:

```
Tag <tab> TWOPARAM <tab> BOOL <tab> REAL1 <tab> REAL2 <tab> Prompt <tab> Prompt1 <tab> Prompt2
```

**USAGE:**

```
Twoparam.Printl(Tag)
    Tag  STRING  Optional. Prints a different Tag than the
              one used in the call to TWOPARAM.New
```

## Twoparam.Check()



### Related Topics

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Copy the current value of the BOOL embedded in a Twoparam into a BOOL.

**USAGE:**

```
Value = Twoparam.Check()
    Value  BOOL  Current value of the Twoparam's BOOL.
```

## Twoparam.SetV1()



### Related Topics

[TYPE:](#) INSTANCE FUNCTION

[PURPOSE:](#) Set the current value of the first REAL embedded in a Twoparam. Returns the value which was just set.

[USAGE:](#)

```
Twoparam.SetV1(Value)
Value    REAL    Value to set the Twoparam's first REAL
```

## Twoparam.SetV2()



### Related Topics

[TYPE:](#) INSTANCE FUNCTION

[PURPOSE:](#) Set the current value of the second REAL embedded in a Twoparam. Returns the value which was just set.

[USAGE:](#)

```
Twoparam.SetV2(Value)
Value    REAL    Value to set the Twoparam's second REAL
```

## Twoparam.V1()



### Related Topics

[TYPE:](#) INSTANCE FUNCTION

[PURPOSE:](#) Copy the current value of the first REAL embedded in a Twoparam into a REAL.

[USAGE:](#)

```
Value = Twoparam.V1()
Value    REAL    Current value of the Twoparam's 1st REAL
```

## Twoparam.V2()



### Related Topics

[TYPE:](#) INSTANCE FUNCTION

[PURPOSE:](#) Copy the current value of the second REAL embedded in a Twoparam into a REAL.

[USAGE:](#)

```
Value = Twoparam.V2()
Value    REAL    Current value of the Twoparam's 2nd REAL
```



## TWOPARAM.Dialog



*Related Topics*

TYPE: LICENSE

PURPOSE: Allow a Twoparam to be edited, saved, and restored in Setup(). The Twoparam will be edited using the fields:

Prompt [X] On(or Off) Prompt1 Val1 Prompt2 Val2

Selecting the checkbox changes the state of the BOOL, changes the label after the checkbox, and grays or ungrays the rest of the object's additional prompts and values. When the checkbox shows a check mark, the BOOL is TRUE, the label is "On" and the text is black. When the checkbox is not checked, the BOOL is False, the label is Off, and the text is grayed out and cannot be edited.

When the operator enters a value into Val1 or Val2, Setup will attempt to turn it into a valid REAL and replace the REAL in Twoparam with it. For example, if an operator puts in "1", Setup() will change it to "1.0". If Setup() cannot translate a string into a REAL, it will default to the value 0.0.

## VCONST



*Related Topics*

TYPE: CLASS

PURPOSE: The VCONST class describes a constant voltage waveform which can be applied by a potentiostat/galvanostat/zra while in the potentiostat mode. The Vconst object encapsulates information about the applied voltage, the elapsed time, and the data acquisition rate. Most of the activity associated with the Vconst happens in the background while a Cpiv.Run() is being executed.

## VCONST.New()



*Related Topics*

TYPE: CLASS FUNCTION

PURPOSE: Create a new constant voltage signal generator.

USAGE:

```
Vconst = VCONST.New(Tag, Pstat, Voltage, Time, SampleTime)
```

Tag	STRING	Object tag.
Pstat	PSTATCLASS	Potentiostat which will use the signal.
Voltage	REAL	Applied voltage in volts
Time	REAL	Total Time in seconds.
SampleTime	REAL	Time between data acquisition steps.
Vconst	VCONST	The object created.

## Vconst.Tweak()

**Related Topics**

**TYPE:** INSTANCE FUNCTION

**PURPOSE:** Change the parameters of an existing constant voltage signal generator.

**USAGE:**

Vconst.Tweak(Pstat, Voltage, Time, SampleTime)

Pstat	PSTATCLASS	Potentiostat which will use the signal.
Voltage	REAL	Applied voltage in volts.
Time	REAL	Total Time in seconds.
SampleTime	REAL	Time between data acquisition steps.
Vconst	VCONST	The existing Vconst object.

## VectorCount()

**Related Topics**

**TYPE:** REGULAR FUNCTION

**PURPOSE:** Returns the dimension of a VECTOR object.

**USAGE:**

Dimension = VectorCount(Vector)

Dimension	INDEX	The maximum number of elements in Vector.
Vector	VECTOR	A VECTOR object.

## VectorNew()

**Related Topics**

**TYPE:** REGULAR FUNCTION

**PURPOSE:** Generate a new VECTOR. A VECTOR is a linear array that can hold any Explain datatype or object.

The number, but not the type, of the VECTOR's elements is specified when the VECTOR is created.

**USAGE:**

Vector = VectorNew(Number)

Vector	VECTOR	The new VECTOR.
Number	INDEX	The maximum number of elements in the VECTOR.

## Warning()



*Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Display a warning message for the operator. Operator must key **OK** before the script will continue.

USAGE:

Warning(Message)

Message STRING The text in the Warning box.

## WinExec()



*Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Start execution of a Windows application by a call from Explain. Equivalent to a Program Manager **File, Run...** command.

USAGE:

WinExec(Command, Mode)

Command STRING The pathname of the program to run.

Mode INDEX A code describing the Window in which the program will run and the program status.

0 = hidden

1 = activate, show normal size

2 = activate, show minimized (as icon)

3 = activate, show maximized

(full screen)

4 = don't activate, show normal size

5 = activate, show at current size

7 = don't activate, show minimized

NIL same as INDEX = 7

## Yield()



*Related Topics*

TYPE: REGULAR FUNCTION

PURPOSE: Allow other Windows applications to run, by temporarily suspending the Explain script. Control will return to the Explain script when Windows has nothing else to do.

This function is useful in preventing a tight loop from hanging up Windows. Simply place a Yield() call in the loop. Obviously, time critical operations should not call Yield().

Sleep() and Dawdle() also give Windows time to execute other programs.

USAGE:

Yield()

## Auto Scripts: Introduction

Auto scripts allow you to automate your experiments. You can schedule experiments to run at specific times and run repeatedly.

To use the auto scripts requires use of two special types of Explain scripts, the "master" script and the "auto" script. A master script is simply an Explain script which 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 which, in turn, run the experiments. The master script provides a way to schedule the timing and repetitions of the auto scripts.

Auto scripts are simply experiments designed to be launched by a master script. They are generally designed to be run unattended although this is not mandatory.

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 in 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 doesn't 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 from the auto script. A side effect is that you can't 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 we've added to our application packages. Both original and automated versions create the same data file formats so the Gamry Echem Analyst will work with data files from either.

## Sequence Scripts: Master Scripts

Master scripts are simply Explain scripts which call other scripts using the LaunchWait() function. The master script is always a custom script. The simple script below just runs a series of different experiments, one after another:

```
include "explain5.exp"
include "Auto Utilities.exp"

function Main()

  if (LaunchWait("Auto Corrosion Potential.exp","auto.set","CPOT","autocpot.dta",1,NIL) eq
FALSE)
    return
  if (LaunchWait("Auto Polarization Resistance.exp","auto.set","RP","autorp.dta",1,NIL) eq
FALSE)
    return
  if (LaunchWait("Auto Potentiostatic EIS.exp","auto.set","EIS","autoeis.dta",1,NIL) eq FALSE)
    return

  Notify("All Experiments Done")
  Dawdle()
```

## How Master Scripts and Auto Scripts interact

Auto scripts are called from a master script using the LaunchWait() function. Suppose you put the following line in a master script:

```
if (LaunchWait("AUTO CORROSION POTENTIAL.EXP","CORPOT.SET","SETUP1","AUTOCPOT.DTA",1,3) eq FALSE)
  return
```

LaunchWait() will start the script "AUTO CORROSION POTENTIAL.EXP" and pass it the following information:

SetupFile	"CORPOT.SET"
SetupName	"SETUP1"
OutputFile	"AUTOCPOT.DTA"

```
PstatNo      1
MuxChannel   3
```

The script AUTO CORROSION POTENTIAL.EXP does the following:

1. It first creates a runner window.
2. It then opens CORPOT.SET and loads the parameters from SETUP1. This is where you would normally see the Setup Dialog.
3. Next it will open the output data file, AUTOCPOT.DTA and acquire use of the hardware, Pstat1 and Mux1.
4. It will then perform the corrosion potential test on channel 3 of Mux1.
5. Finally, it will close everything and end. Upon ending, it returns a status value to the LaunchWait() function in the master script which has been waiting.
6. The master script can tell if AUTO CORROSION POTENTIAL.EXP ran OK by testing LaunchWait()'s return value. In the example above, the master script terminates if AUTO CORROSION POTENTIAL.EXP didn't run correctly for any reason.

## Using the Auto Scripts - A Thumbnail Sketch

To schedule a series of experiments you need to do the following:

- Create a series of setup files containing the experiment parameters you want to use.
- Create a master script listing the auto scripts and times you want them to run.
- Set up your electrochemical cell and run the master script.

## Creating an Experiment Setup

To create the setup files you must run the corresponding original experiment. In the Setup dialog select the parameters you want to run and use the **Save** button in the Setup dialog box to store the experimental setup information on disk. You then use the **Cancel** button to avoid actually running the experiment. Later, you will tell the auto version what setup name to pull off the disk.

For example, you might want to run a polarization resistance experiment as part of a sequence of experiments. Use the Gamry Framework and call up the polarization resistance experiment. After you have set the experimental parameters in the Setup dialog box click the **Save** button to store the parameter set. Look for "Experimental Parameters and Parameter Sets" in the DC105 or EIS300 manual tutorial sections if you need additional detail on saving setups. You can name the setup file and parameters anything you like, for example "MYPARMS.SET" and "MyRpSetup". You will need these names later when editing the sequencer script.

## Creating a Master Script

The easiest way to create a master script is to use the Gamry editor to modify one of the sample master scripts we've included. The sample master script above is named RUN MANY.EXP and is included in the Gamry Framework installation. You can load RUN MANY.EXP into the Gamry Framework by using the File/Open command. To avoid actually modifying RUN MANY.EXP, immediately use File Save As to change the filename to a new name like MYEXPS.EXP. You can now edit this file without worrying about changing RUN MANY.EXP. This sample script carries out three experiments, a Corrosion Potential experiment, a Polarization Resistance experiment and an Electrochemical Impedance experiment. Each experiment is run by using calls to LaunchWait(). The NIL's in the place of MuxChannel indicate that no multiplexer is used.

Look again at the first line in RUN MANY.EXP that starts an auto script:

```
if (LaunchWait("Auto Corrosion Potential.exp", "auto.set", "Ecorr", "autocpot.dta", 1, NIL) eq
FALSE)
  return
```

The first parameter to the first LaunchWait() is AUTO CORROSION POTENTIAL.EXP. This is the name of the script file that will run the Corrosion Potential experiment.

Suppose you need to set up a polarization experiment in your master script. You would need to list AUTO POLARIZATION RESISTANCE.EXP as the first parameter to LaunchWait(). Continuing on, suppose you are using the setup created above (MyRpSetup in MYPARMS.SET) and you want to use potentiostat 1 on multiplexer channel 5 and save the data into RP.DTA. You would edit the first call to LaunchWait to read:

```
if (LaunchWait("Auto Polarization Resistance.exp", "myparms.set", "MyRpSetup", "rp.dta", 1, 5) eq
FALSE)
  return
```

Note that it doesn't matter whether you refer to "AUTO POLARIZATION RESISTANCE.EXP" or "auto polarization resistance.exp" or "Auto Polarization Resistance.Exp". The strings used when calling LaunchWait() are not case sensitive.

You should be careful about the names of the parameter setup and setup file. If you mistakenly give a non-existent name for the file or the

setup, your master script will give you an error message "SetupRestore failed", pause until the message is acknowledged, then abort that experiment script. You will then be given the option of aborting all experiments or continuing on to the next experiment.

Another place you should be careful is in naming the output data file. If this file exists beforehand, it will be overwritten when that particular experiment is run. You will not be given any warning messages about an existing file.

The multiplexer number used in the experiment scripts is always assumed to be the same as the potentiostat number. For example, if a multiplexer is used with potentiostat 1 it is assumed to be multiplexer 1, while potentiostat 2 is paired with multiplexer 2. If you are not using a multiplexer, just use NIL as the channel number.

The RUN MANY.EXP master script is designed to run 3 experiments sequentially. If you don't need that many experiments, you can delete the extra lines that call LaunchWait(). Alternatively, if you need more than 3 experiments you can use the Copy function of the Gamry editor to add extra calls to LaunchWait().

## Controlling Timing

The experiment timing can be controlled using the WakeUp() and Delay() functions. WakeUp() is used to control the start time of an auto script.

For example:

```
if (WakeUp("15:30:00",NIL) eq FALSE)
  return
if (LaunchWait("Auto Polarization Resistance.exp", "myparms.set", "MyRpSetup", "rp.dta", 1, 5) eq FALSE)
  return
```

In this example the function WakeUp() sleeps until 'today' at 3:30 pm at which time AUTO POLARIZATION RESISTANCE.EXP is launched. To change the time edit the "15:30:00" string to whatever time you prefer. The times are expressed using a 24 hour clock and the last pair of digits are the seconds setting. Yes, the seconds setting is needed. The NIL parameter is just a shorthand way of saying use today for the date. If you need to start your experiment at a later date replace the NIL parameter with the appropriate date string like "06/25/99".

If the time has already past when the script runs, the WakeUp() returns immediately. For example, if you use Wakeup("15:30:00", NIL) and it is 4:00 PM when the master script is started, the call to WakeUp() will not pause.

Because WakeUp() uses your computer's clock time and date, it is important to check them periodically for accuracy. WakeUp() also pays attention to your Window's International Date & Time Format selections. Please refer to your Windows manual for international information.

If you prefer that the script just pause for a few minutes use the Delay() function. For example:

```
if (Delay(600) eq FALSE)
  return
LaunchWait("Auto Potentiostatic EIS.exp", ...)
```

In this example the master script will delay for 600 seconds and then continue on. If the operator should hit the **Skip** key and then selects "Terminate the script", Delay() returns FALSE and the script "returns". If you hit the **Skip** key and then select "Skip to the next experiment", Delay() returns TRUE and the "Auto Potentiostatic EIS.exp" is launched immediately.

You can control a delay with respect to a fixed point in time by using the DelayFrom() and Time() function. For example:

```
StartTime = Time()
; Do Something Arbitrary In Time Here
; ...
if (DelayFrom(600,StartTime) eq FALSE)
  return
```

In this example the StartTime is memorized. DelayFrom() will delay until 600 seconds after StartTime.

## Other Ways to Launch an Auto Script

We have created a few variations on LaunchWait(), all taking the same parameters.

Launch(AutoScriptFile, ...) will start AutoScriptFile and return immediately. This is useful if you want to run multiple auto scripts at the same time. Be careful that they don't use the same potentiostat since only one script may use a given potentiostat at any given time.

Launch() and LaunchWait() print information about the currently launched auto script and allow the operator to intercede in case the auto

script fails. There are two similar functions, `Execute()` and `ExecWait()`, which perform no tasks other than to simply launch the auto script.

## Running a Master Script

Once you are done editing your master script, save it to disk using **File/Save**.

You may want to test your master script before you use it by running the experiments on a simple dummy cell. This will help you locate any typos or syntax problems without ruining a specimen.

When you have your cell set up, you can use **Ctrl+R** to run the current script. If you are reusing a master script and it is not loaded, click on the Experiment menu and then select the Named Script option. The Gamry Framework will ask you for the name of the script to run. Type in the name of your master script like the MYEXPS.EXP file that we created above.

When your master script is loaded, a new window will open and the script will start to run. This runner window will display messages about what experiment is currently running or how much time is left in a time delay. When an experiment is launched and running, a second window will open for that experiment. This experiment window will look and act like a normal experiment window but at the end of the experiment it will close automatically.

If you wish to terminate an auto script in the middle of an experiment, select **F1-Abort**. After the experiment terminates the master script will ask whether or not you wish to abort all experiments or just continue with the next experiment.

Pressing **F1-Abort** from the master script causes the master script to terminate immediately. Any running auto scripts will continue to run unless you terminate them explicitly. If you wish to skip the delay in a master script, use the **F2-Skip** button to continue on to the next experiment immediately. The master window will update periodically to reflect what experiment is currently running. When the master script is finished running, the master window will display "All Experiments Done".

## How to Repeat Experiments

If you wish to repeat one experiment a certain number of times, you can copy one call to `LaunchWait()` as many times as needed. Although this technique will work for small numbers of experiments, a better solution for large numbers of experiments is to create a loop to repeat that experiment.

A sample master script has been created to repeat one experiment multiple times. You will find it in the file REPEAT1.EXP.

```
include "explain5.exp"
include "Auto Utilities.exp"

function Main()

  if (WakeUp("15:30:00",NIL) eq FALSE) ; NIL implies Today.
    return
  BaseName = "rpdata"
  i = 1
  Cycles = 5
  while (i le Cycles)
    FileName = Sprint(BaseName,i, ".dta")
    if (Delay(10) eq FALSE)
      return
    if (LaunchWait("Auto Polarization Resistance.exp", "auto.set", "Rp Setup", FileName, 1, NIL)
    & eq FALSE)
      return
    i = i + 1
  Notify("All Experiments Done")
  Dawdle()
```

You can modify REPEAT1.EXP. Use **File, Open** to edit it, but remember to save your changes under a new script name using the **File, Save As** command.

REPEAT1.EXP creates the output filenames automatically by combining a base filename with a loop number and a ".dta" file extension. In this case the master script is running a polarization resistance experiment 5 times with the base filename of "rpdata", so the output files are named RPDATA1.DTA thru RPDATA5.DTA. To change the output filenames edit the line where the script sets BaseName to "rpdata". You can change the number of times the experiment is run by changing the value of Cycles.

Since the base filename is combined with a cycle number to generate the output filename, you need to be careful that the base filename plus

the total number of cycles is less than 256 characters.

The call to Delay() controls how much time passes between the end of one experiment and the start of the next, while LaunchWait() controls the actual start time of the experiment. Edit these lines to fill in your own preferences for the time delay and the actual experiment run. The only precaution here is to avoid accidentally replacing the FileName parameter in the call to LaunchWait(). The other parameters are used the same way as in RUN MANY.EXP.

From a beginner's standpoint, this script is more complicated than RUN MANY.EXP but it does illustrate some of the flexibility that the Explain scripting language gives you. For example if you want to run three different experiments followed by 10 repetitions of one experiment followed by 2 different experiments and then another loop, you could easily create a new master script by copying portions of RUN MANY.EXP and REPEAT1.EXP as needed to set up your particular experiment sequence. You can also do anything in a master script that you can do in a regular script such as opening a file, writing notes, etc.

## Adapting Custom Experiment Scripts To Auto Scripts

You don't need to read this section unless you have a custom experiment script that you would like to get set up to run with a master script. As mentioned in the previous help sections, any experiment script run with LaunchWait() should be designed to run completely unattended. What do you do if you have modified one of our standard scripts to create a custom experiment? If your modifications are small and we supply an automated experiment script that is similar to your modified script, you can examine our automated script and modify it to suit your needs. Most of the code is identical in both the original and automated versions.

If you have created a completely new experiment script and need to modify it, you should look at how the scripts "Auto Polarization Resistance.exp" and "Polarziation Resistance.exp" compare. Due to their length it is not practical to reproduce them here and point out their differences line by line. (Each one is roughly 3 typed pages in length.) These two scripts do, however, illustrate the differences between a normal experiment script and its automated version.

The key differences are:

- Parameters are passed into Main() using a vector of values.
- SetupRestore() is used instead of Setup()
- Error codes are returned from Main()
- Functions requiring user input are avoided, e.g. Warning() or Query().
- Output.Open(TRUE) is used to avoid user input.
- If the multiplexer is used, the file "MUXLOAD.EXP" is included and ChannelNo is tested before all multiplexer calls.

Please contact our office if you require additional assistance.

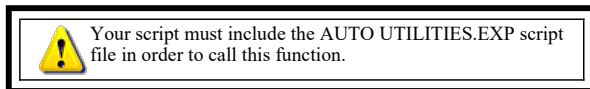
## Delay()



### Related Topics

**TYPE:** REGULAR FUNCTION

**PURPOSE:** Sleep for a number of seconds. Operator can use the Skip (F2) button to wake up early.



**USAGE:**

Status = Delay(Seconds)

Minutes	INDEX	Number of seconds to delay
Status	BOOL	TRUE     Woke up OK or Operator selected SKIP and Continue
		FALSE    Operator selected SKIP and Terminate.

## DelayFrom()





### Related Topics

**TYPE:** REGULAR FUNCTION

**PURPOSE:** Sleep for a number of minutes after a fixed point in time. Operator can use the skip button to wake up early.



Your script must include the AUTO UTILITIES.EXP script file in order to call this function.

**USAGE:**

Status =

Status = DelayFrom(Minutes, StartTime)

Minutes	INDEX	Number of minutes to delay
StartTime	INDEX	Reference time
Status	BOOL	TRUE    Woke up OK or Operator selected SKIP and Continue FALSE    Operator selected SKIP and Terminate.

## LaunchWait()



### Related Topics

**TYPE:** REGULAR FUNCTION

**PURPOSE:** This function loads and runs another script.



Your script must include the AUTO UTILITIES.EXP script file in order to call this function.

LaunchWait() is defined in AUTO UTILITIES.EXP and simply calls ExecWait(). The Launch() command updates the master script window before calling the corresponding Exec() function. It also deals with error returns a bit more civilly.

Normally you would call LaunchWait(). Use LaunchWait() when you want to run two sequential experiments on the same potentiostat.

**USAGE:**

Result = LaunchWait(ScriptFile,SetupFile,SetupName,OutputFile,PstatNo,ChannelNo)

ScriptFile	STRING	Name of script file to launch.
SetupFile	STRING	Name of file containing setup parameters.
SetupName	STRING	Name of the specific setup in the setup file to be used.
OutputFile	STRING	Name of the output file for saving experimental data.
PstatNo	INDEX	Number of the potentiostat and multiplexer to be used(1-4).
ChannelNo	INDEX	Multiplexer channel to use. NIL implies either no multiplexer or use default channel setting.
Result	BOOL	TRUE if experiment launched & ran okay FALSE if aborted or a problem occurs.

## WakeUp()



**TYPE:** REGULAR FUNCTION

**PURPOSE:** Sleep until a preset time and date. Operator can use the skip button to wake up early.

Your script must include the AUTO UTILITIES.EXP script file in order to call this function.

**USAGE:**

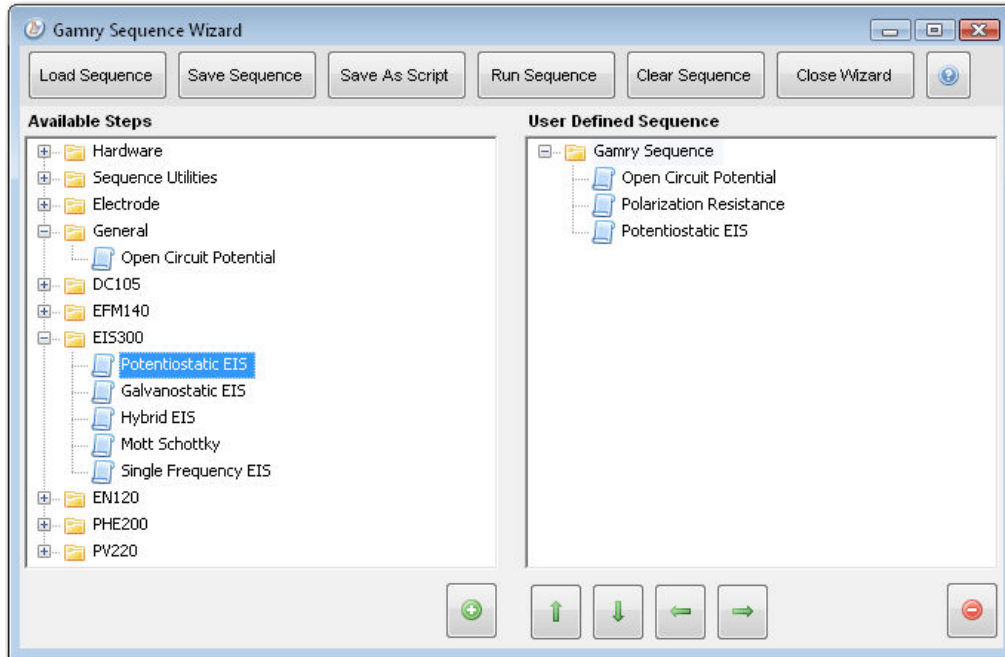
Status = WakeUp(TimeString, DateString)

TimeString	STRING	Time in format HH:MM:SS. Windows Control Panel can be used to set non-US formats.
DateString	STRING	Date in format MM/DD/YY. Windows Control Panel can be used to set non-US formats.
	NIL	Today
Status	BOOL	TRUE            Woke up OK or Operator selected SKIP
		FALSE          Operator selected Abort

## Sequence Wizard - Introduction

The Gamry Sequence Wizard is a tool to help end users create custom experiments. The Sequence Wizard is not the same as the auto scripts. The wizard is designed to allow a point and click approach to sequencing experiments. This simple approach should make even the most daunting sequence easy to attain. Drag and drop accessibility makes for easy re-ordering of sequence steps. Variables, loops, and delays make for a flexible sequence tool.

The image below contains hot-links. Find out more about different aspects of the Sequence Wizard by clicking on the image in different locations.

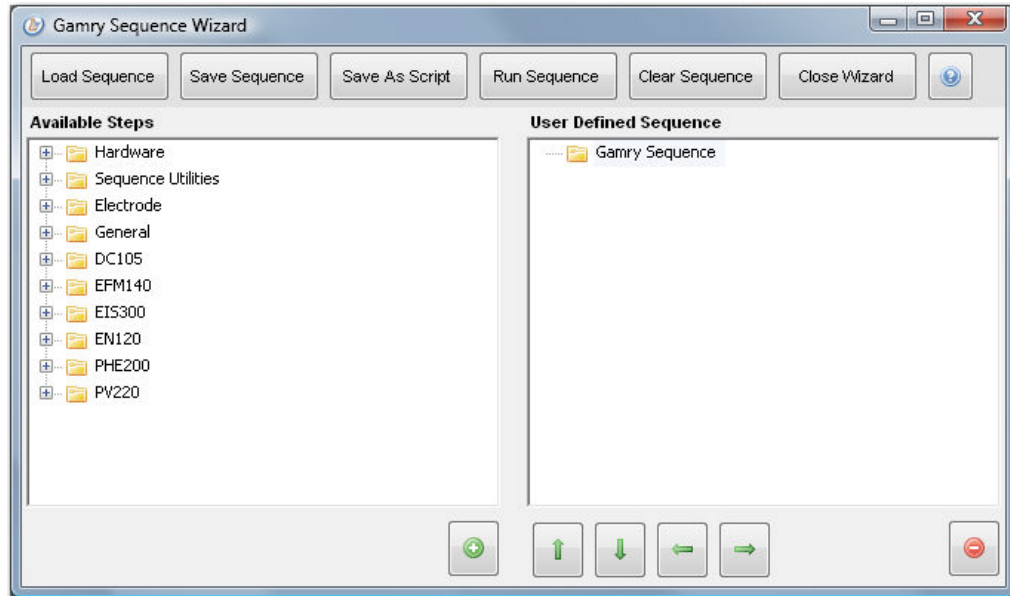


Once you are comfortable with the user interface, continue on to [Your First Sequence](#).


## Your First Sequence

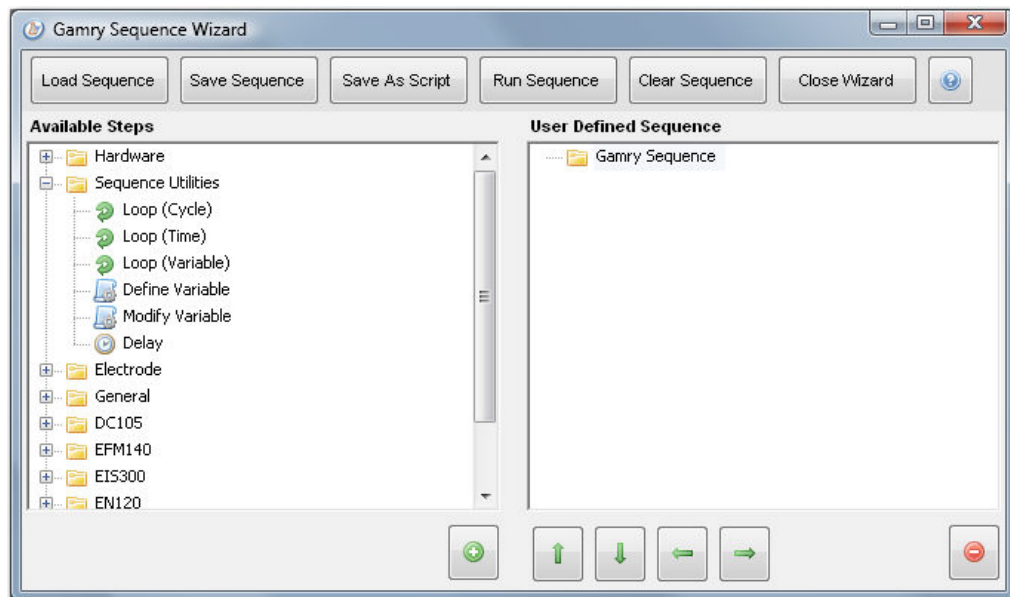
Creating a sequence is now as easy as using the mouse to select your steps. The following tutorial will help you to create your first sequence. When you are finished, this sequence will loop a 20 second Open Circuit Potential experiment 3 times, with a 10 second pause between Runs. Now on to the sequence!

1. Start the Gamry Framework by clicking on the Framework icon.
2. Bring up the **Sequence Wizard** by selecting Sequence Wizard from the **Experiment** Menu of the Framework or pressing (**CTRL + W**).
3. You should now see the Gamry Sequence Wizard window which looks like the following:

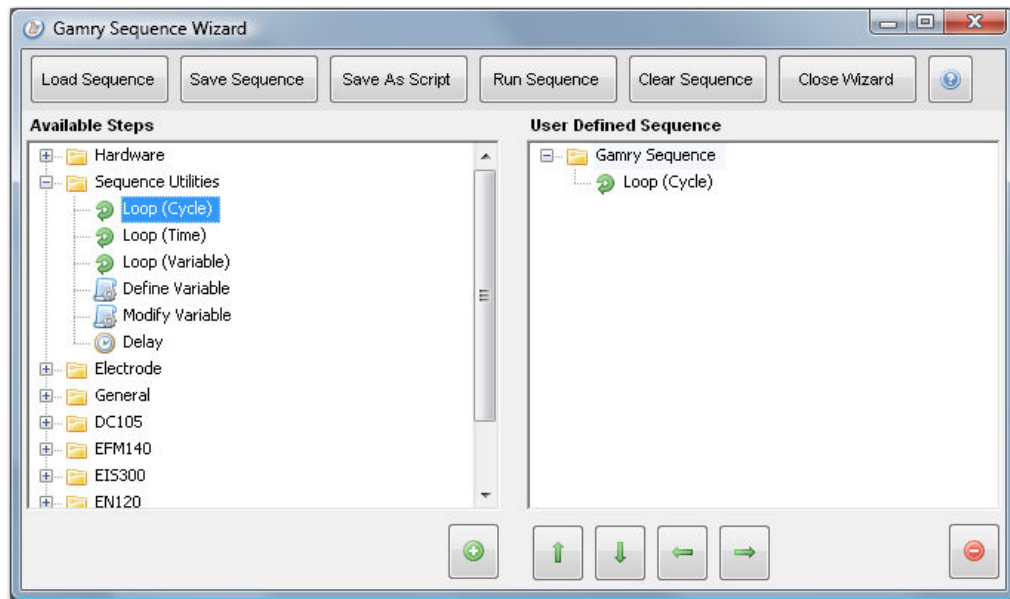


You may or may not have all of the categories listed under the *Available Steps* tree that are pictured above. Do not worry, as only the packages you purchased show up in this tree.

4. Expand the Sequence Utilities category by clicking on the plus sign, , next to Sequence utilities. Your window will then look similar to the following:

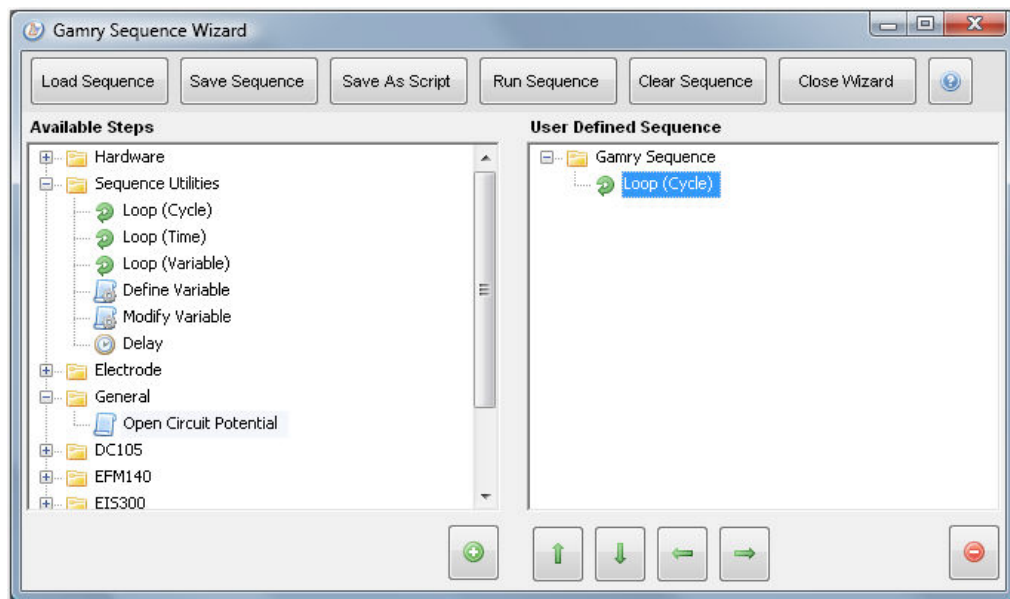


5. Now double click the Loop (Cycle) icon beneath the in the Available Steps tree. You should then see it appear beneath the User Defined Sequence. If you prefer you can simply single click the Loop (Cycle) icon and then click the Add to Sequence Button instead. Either way, your window should look similar to the following:

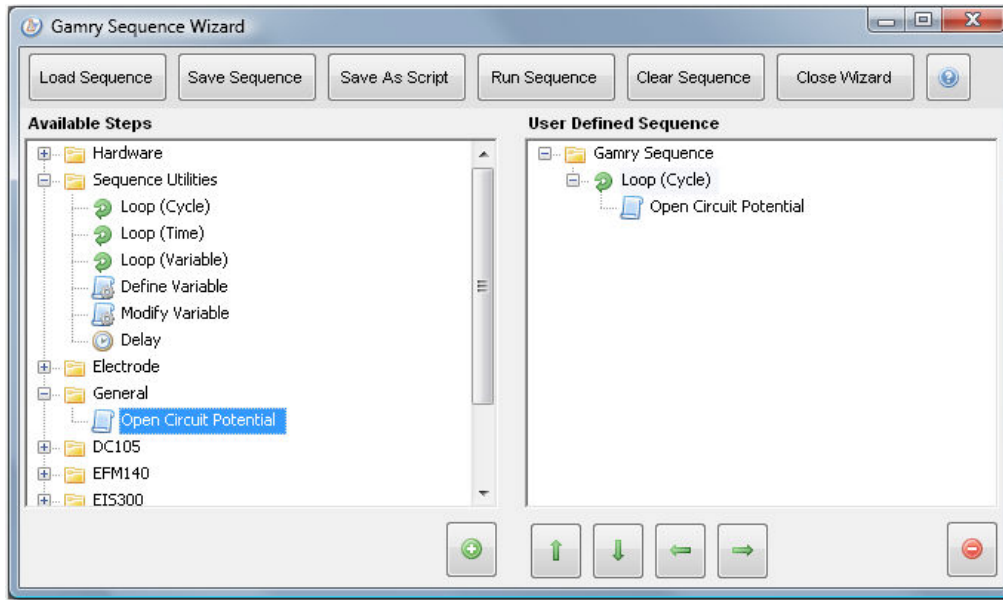


We have just added our first step to the User Defined Sequence. Now a loop by itself is not really useful, so we should add something underneath it.

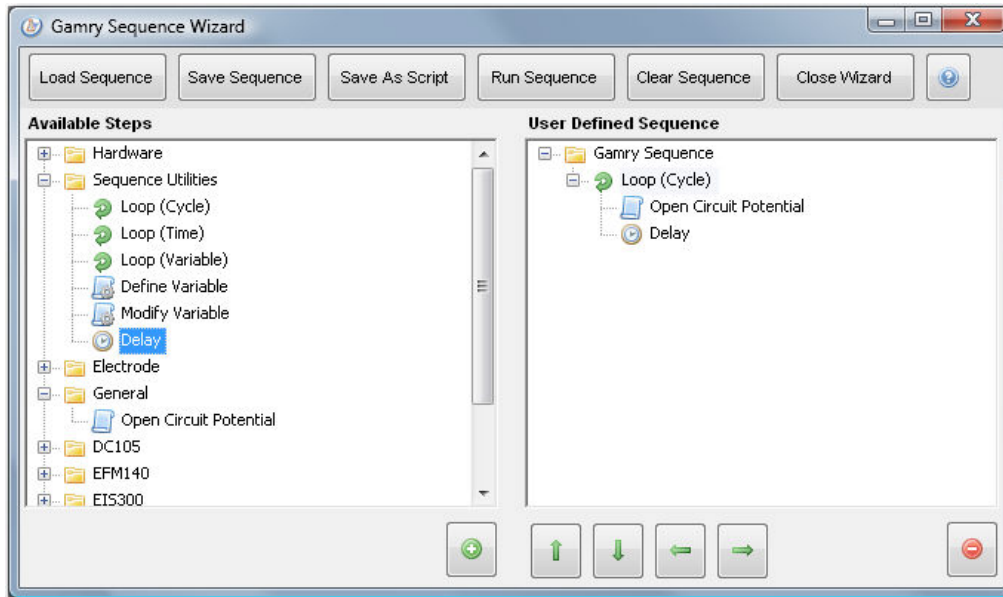
6. The current selection in the User Defined Sequence is Gamry Sequence. Click on the Loop (Cycle) icon beneath the User Defined Sequence. This will make it the current selection as shown below:



7. Now expand the General Category under the Available Steps tree and double click (or single click followed by clicking on the Add To Sequence Button) the Open Circuit Potential step. This will add it to the User Defined Sequence as a child of the Loop. Children of the loop are the steps that are actually looped. If we skipped step 6, the Open Circuit Potential step would not have been added as a child of the Loop. Your sequence should now look like:



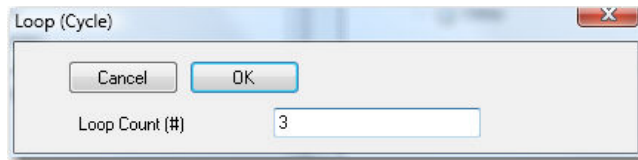
- Now lets add one more step to our sequence. Add the Delay step from under Sequence Utilities by double clicking it. It will be placed beneath the Open Circuit Potential step. Your sequence will now look like:



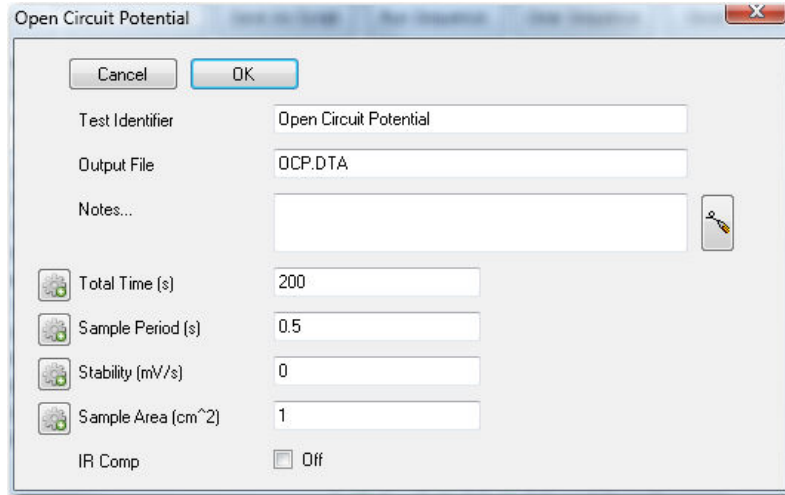
- Now we have the basis for our sequence. Lets set some parameters for each of the steps. We set parameters by double clicking on the step beneath the User Defined Sequence, or by right clicking on a step and selecting **Properties**. Lets start by checking the parameters of our Loop. If we double click the Loop icon under the User Defined Sequence, we will get a dialog box similar to the one shown below:



Now we can edit the Loop Count. Lets make it 3. Then press OK. If you were to again check the properties by double clicking on the Loop step you would see that it shows we have a Loop Count of 3.

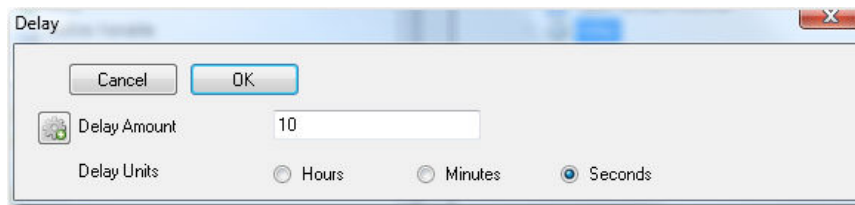


10. Now let's check our Open Circuit Potential parameters by double clicking on the Open Circuit Potential step in our User Defined Sequence tree. You should see a dialog box similar to the following:




Let's shorten the Total Time from 200 seconds to just 20 seconds. After you have done this, simply select OK.

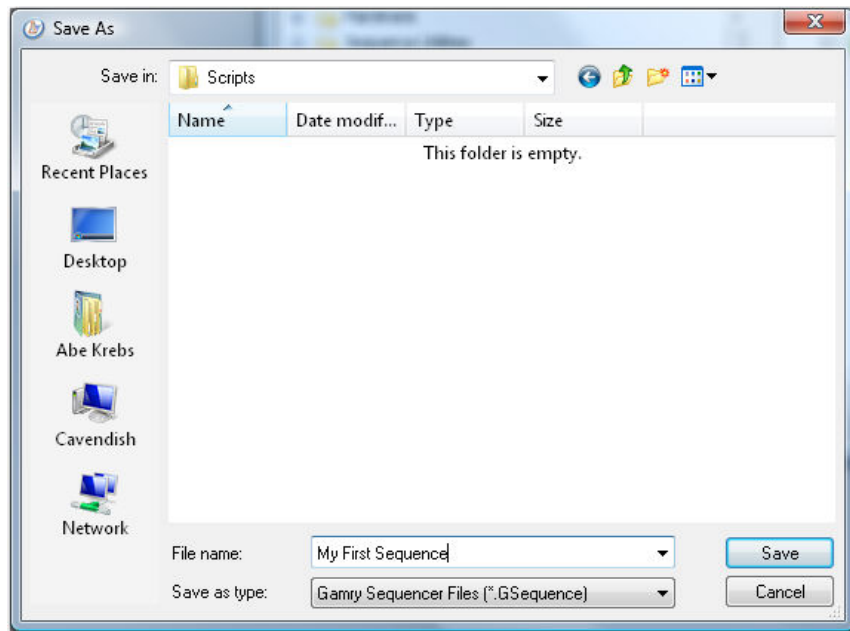
11. Next let's check our Delay parameters by double clicking on the Delay step in our User Defined Sequence tree. Your dialog should be similar to the one shown below:




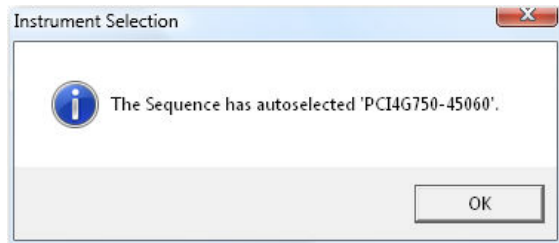
It defaults to a 10 second delay. This is good enough for our needs, so we can just leave it alone and press OK.

12. Now, before we do anything else, let's save our sequence. We do this by clicking on .

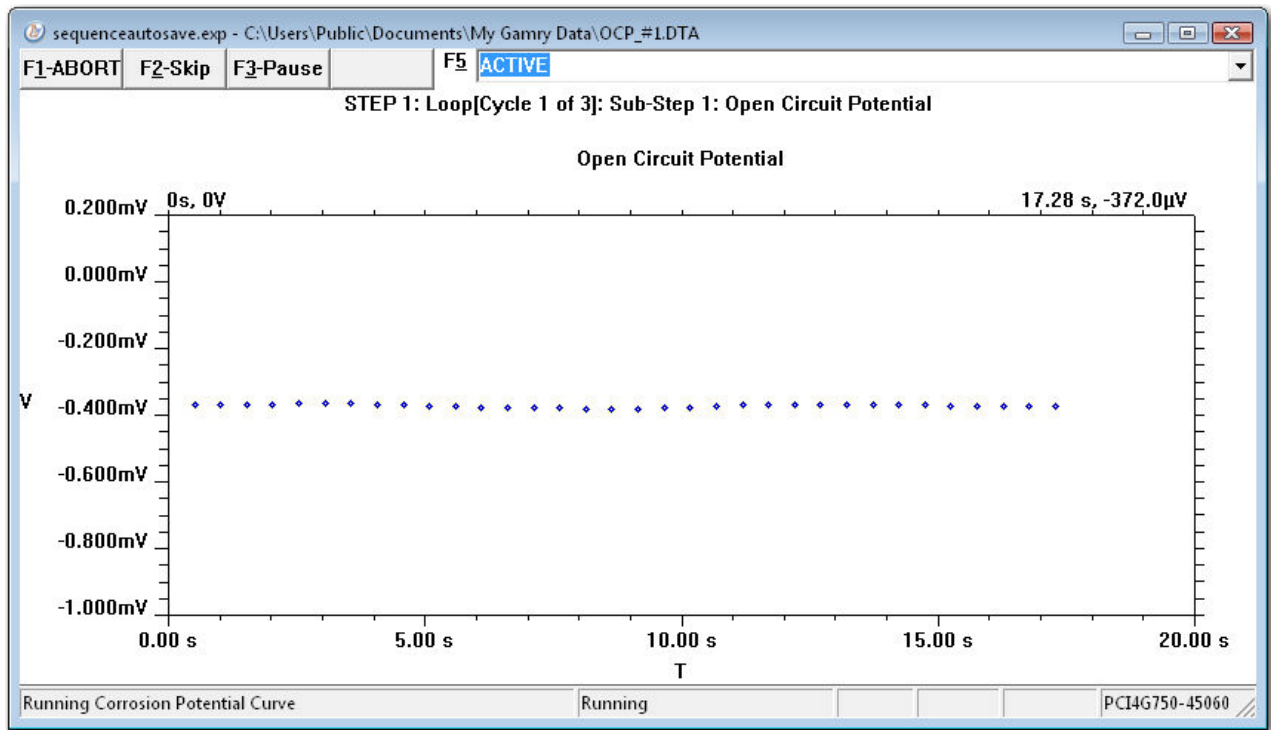
You will get a dialog box where you can enter a name. Enter something like *My First Sequence*, as shown below:



13. Finally, we are ready to run. Select  to generate and run the sequence.
14. When the script begins, it will select the first available potentiostat and tell you what instrument it selected with a dialog box like the one shown below:



15. After pressing OK, the sequence will begin to run. For this purpose we will assume you have never run this sequence before, so it will generate new data files for you, OCP\_#1.DTA, OCP\_#2.DTA, and OCP\_#3.DTA, one for each of the loop cycles. If this sequence has been run before and the data files already exist, you will be asked if you wish to overwrite them.
16. When the sequence is running, you will see a window similar to the one below:



Notice in the Headline area at the top it states the following:

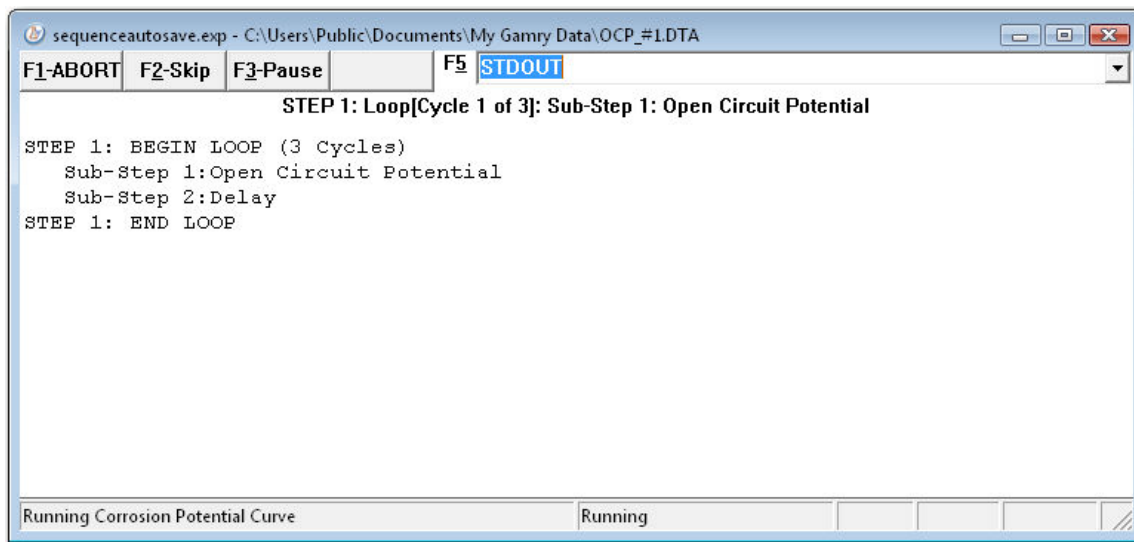
**STEP1: Loop[Cycle 1 of 3]: Sub-Step 1: Open Circuit Potential**

This area will always show what step in the sequence is currently being run. Because the first step is a loop, it then shows the current cycle number as well as the sub-step being run. Our test sequence has 1 step, the *Loop*, with 2 sub-steps, *Open Circuit Potential* and *Delay*. Our *Loop* is to run for 3 cycles.

- Now, lets take a quick look at the STDOUT window by selecting STDOUT from the dropdown list in the runner window (or by pressing F5 and then selecting STDOUT).



You will then see a description of our sequence in the STDOUT window. The window should look like the one shown below:



Notice that the Headline is still visible even though you are looking at the STDOUT window. This makes for easy reference as to your location within the sequence.

- When the sequence is finished running, the Runner window will close automatically.

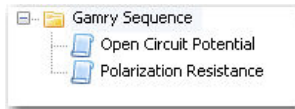


19. Congratulations, you have made your first sequence.

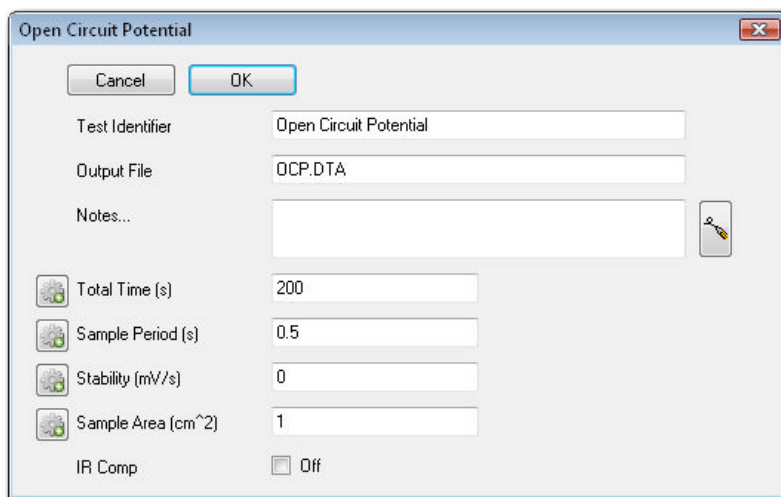
## Entering Parameters

Entering parameters for steps in the Sequence Wizard is very similar to the way they are entered when running a normal Framework script. First, select a step in the User Defined Sequence, and then double-click or right click and select **Properties** to bring up the parameter dialog box.

Let's take a look at the following sequence:



To edit the parameters for *Open Circuit Potential*, double-click on it or right click on it and select **Properties**. You will then be presented with a parameter dialog box that looks like the following:



Edit the setup parameters by clicking in the appropriate edit boxes and typing your changes. When you are satisfied you can click on the **OK** button at the top of the dialog box. If you wish to cancel your changes, simply select **Cancel** instead.

Now let's take a look at the Polarization Resistance setup parameters. Again, simply double-click on the step or right click and select **Properties**. The parameter dialog box will look like the following:

Again you make your changes by clicking in the appropriate edit boxes and typing your values. Press **OK** when you are finished.

If you are wondering what the icon to the left of some of the parameters is, read the section on [Using Variables](#).

## Ordering Steps

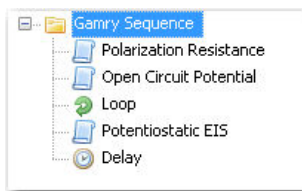
Moving steps around in the sequence is as simple as dragging it to where you want. Alternatively, you can click on the step you wish to move use the ordering arrows



to move it up and down in the sequence, in or out of a loop.

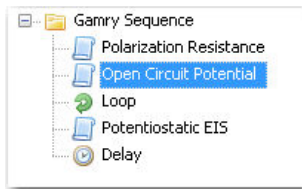
## An Example

Let's take a look at an example. In the sequence below you will see 5 steps, in no particular order:

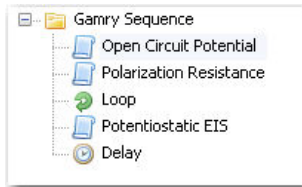



Now this sequence does not make much sense right now, so let's get it the way we want it.

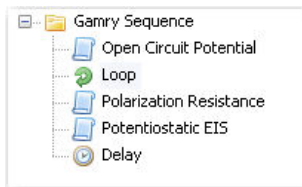
First, we want to have *Open Circuit Potential* be our first step in the sequence. Click on it to select it as shown in the following image:




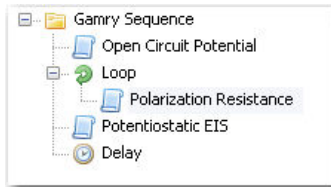
Now click on the Move Up in Sequence Icon  (or press CTRL + UP ARROW). The result will be a sequence that looks like:



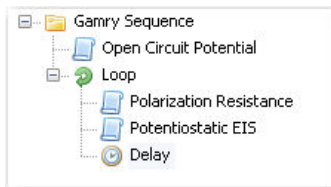
Next, we want to have our *Loop* as the second item. Select it and again use the Move Up in Sequence Icon  to shift it up to the second spot. The sequence will now look like:



Now, we really want the *Polarization Resistance*, *Potentiostatic EIS*, and *Delay* steps to be run in the loop. We need to increase their nest level. Select each one (one at a time) and then click on the Increase Nest Level Icon  (or press CTRL + RIGHT ARROW) to shift it beneath the *Loop* step. Once this is performed for the *Polarization Resistance* step, the sequence will look like:



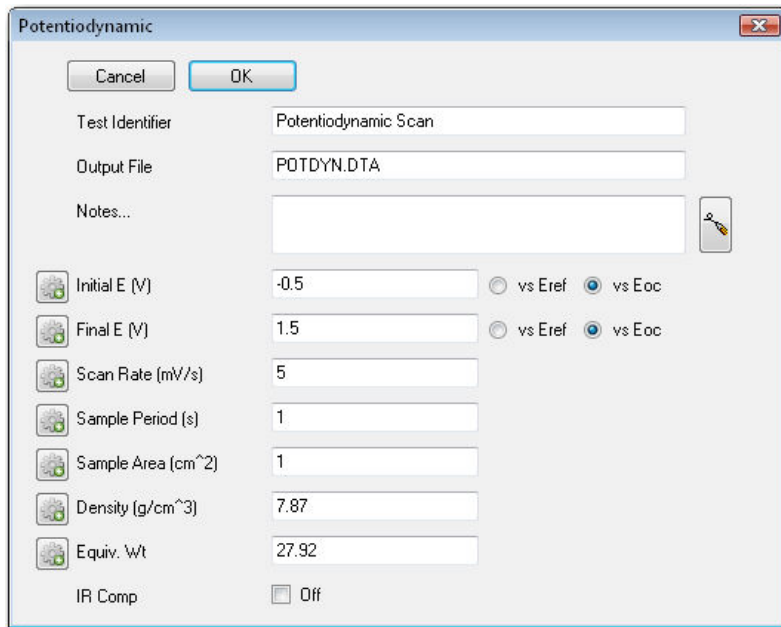
After shifting the other two steps the sequence will ultimately look like:



As you can see, using the moving arrows (or simply dragging items around) you can easily re-order any sequence.

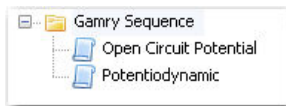
## Data File Names

When running a sequence, it is important to understand how data file names are generated. First, for each step in the sequence that generates a data file, the setup parameters box will have a location to enter a filename. In the picture below, you will see that the Output File name is 'POTDYN.DTA':



## The Simple Example (No Loops)

Now, let's take a look at a quick example. If we run a simple sequence that looks like the following:



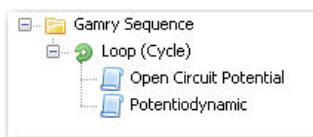
Our two steps have Output files entered like those shown below:



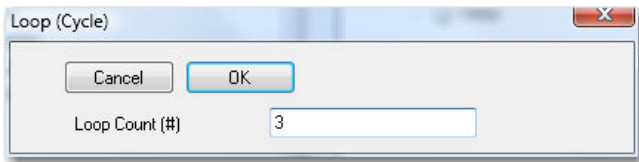
When we run this simple sequence, two data files will be generated, 'OCP.DTA' and 'POTDYN.DTA'. In summary, if no loops exist, the Output File is used as entered in the setup dialog box.

## The Advanced Example (Loops)

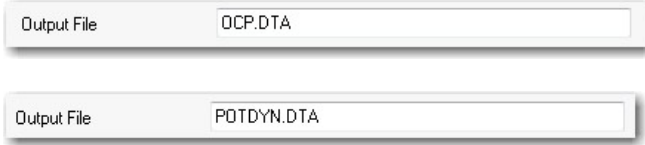
Now, let's take a look at a more advanced example which has a loop. When loops are part of the sequence, the Output File name is used as a base file name, with the loop number being appended to the end. Basically, the trailing .DTA is removed (if it exists at all), an underscore followed by the # sign and the loop cycle is appended, and then the trailing .DTA is placed back. Let's see how it works for the following sequence:



We have entered a Loop Count of 3 as shown below.



The Output files for each of the 2 sub-steps are as they were before:



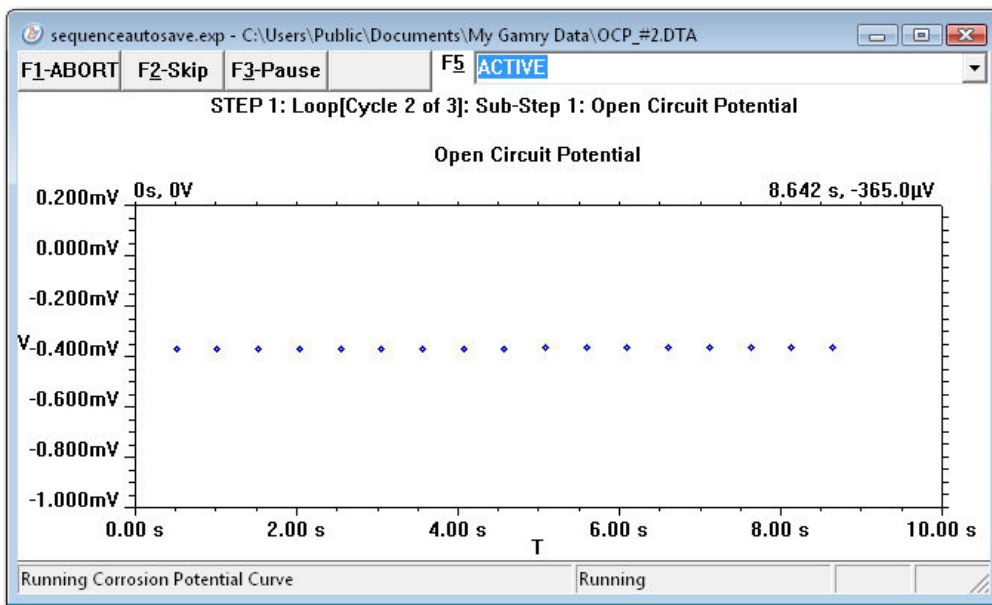
Now, when we run the sequence, the following data files are generated:

For Cycle 1, OCP\_#1.DTA and POTDYN\_#1.DTA

For Cycle 2, OCP\_#2.DTA and POTDYN\_#2.DTA

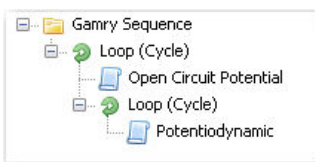
For Cycle 3, OCP\_#3.DTA and POTDYN\_#3.DTA

When the sequence is running, the current data file that is being written to, is always noted in the titlebar as seen in the following image where the current data file is OCP\_#2.DTA:



## The Final Example (Nested Loops)

In the event there are nested loops, an additional cycle number is appended to the data file name in the same manner as when there is only 1 loop. Lets look at the following sequence:



For the outer loop we have a loop count of 3. For the inner loop we have a loop count of 4. Our Output File names are as before:

Output File      OCP.DTA

Output File      POTDYN.DTA

Now, the Open Circuit Potential step is in the outer loop, and will generate filenames like before, OCP\_#1.DTA, OCP\_#2.DTA, and OCP\_#3.DTA. The Potentiodynamic step, however, is now a child of the inner loop. It generates files like the following:

Outer Loop Cycle #1, Inner Loop Cycles 1 to 4, POTDYN\_#1\_#1.DTA, POTDYN\_#1\_#2.DTA, POTDYN\_#1\_#3.DTA, POTDYN\_#1\_#4.DTA.

Outer Loop Cycle #2, Inner Loop Cycles 1 to 4, POTDYN\_#2\_#1.DTA, POTDYN\_#2\_#2.DTA, POTDYN\_#2\_#3.DTA, POTDYN\_#2\_#4.DTA.

Outer Loop Cycle #3, Inner Loop Cycles 1 to 4, POTDYN\_#3\_#1.DTA, POTDYN\_#3\_#2.DTA, POTDYN\_#3\_#3.DTA, POTDYN\_#3\_#4.DTA.

Here you can see that the first appended cycle number belongs to the outer loop, and the second appended cycle number belongs to the inner loop. If you can remember this, you will be able to locate your data files with ease.

## Using Loops

There are 3 different kinds of loops available for use in the sequence wizard. The purpose of a loop step is to re-run a single step, or multiple steps until the loop criteria is met. The different kinds of loops are described below.

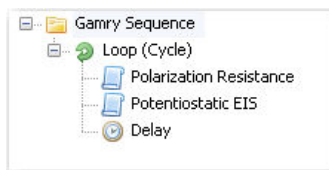
File names are handled automatically by the loop steps, as described by the [Data File Names](#) topic.

Loops are also valuable when [Using Variables](#).

### Loop (Cycle)

The *Loop (Cycle)* step allows you to re-run a single step, or multiple steps for a user specified number of cycles. This step can contain any number of other steps, including other loops. To properly use a *Loop (Cycle)* step, you must first select a *Loop (Cycle)* step from the Available Steps under Sequencer Utilities, and add it to the User Defined Sequence. You then need to add any steps you wish to loop as a child of the *Loop (Cycle)* step. You can do this in 1 of 2 ways. You can select the *Loop (Cycle)* step in the User Defined Sequence, and then add the children from the Available Steps, or you can add all of the steps and then [order your sequence](#) after the fact. Either way, you will end up with the same result.

A simple example of how to use a loop is shown in the following sequence:



You can see in this sequence that there are 3 steps listed as children of the *Loop (Cycle)* step. These child steps (*Polarization Resistance*, *Potentiostatic EIS*, and *Delay*) will run within the loop. By double-clicking on the *Loop (Cycle)* step you can edit the number of cycles the loop will run. The parameter dialog for the *Loop (Time)* step is pictured below:

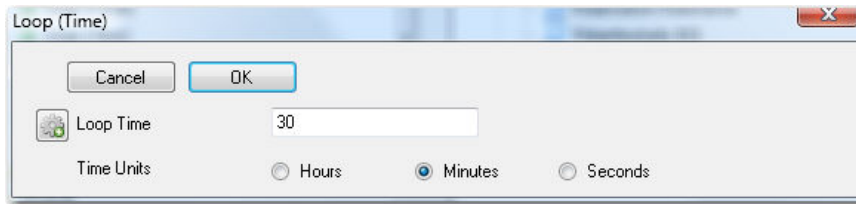


You can see that the default Loop Count is 1. You can change this and then select OK to save your change.

When the sequence pictured above is run, it will first perform a *Polarization Resistance* measurement, followed by a *Potentiostatic EIS* scan. Then the sequence will pause for the user specified time in the *Delay* step. Then the sequence will repeat again, until it has performed all 3 sub-steps for the number of cycles specified by the Loop Count parameter.

## Loop (Time)

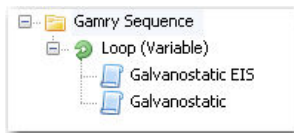
The *Loop (Time)* step is similar to the *Loop (Cycle)* step except it allows you to re-run a single step, or multiple steps for a user specified amount of time. The parameter dialog for the *Loop (Time)* step is pictured below:



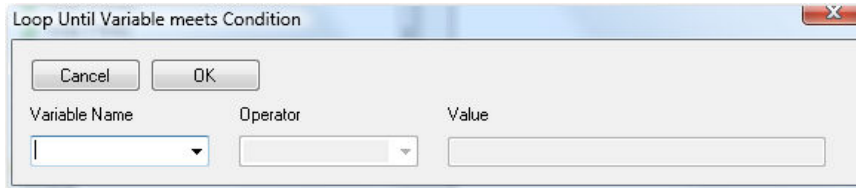
You can see that the default Loop Time is 30 minutes. You can change the Loop Time and Units and then select OK to save your changes. The child experiments will run until the time has expired. Note that ALL child experiments will be completed, as the timing is only checked at the beginning of a loop, not in the middle of the loop. So if the time has not yet expired, another complete loop will be run.

## Loop (Variable)

The *Loop (Variable)* step allows you to loop child steps until a variable meets a certain criteria. Before you use this loop, please read the section on [Using Variables](#). A simple example of using this Loop step is as follows:



Let's say we are connected to a battery. In the sequence pictured above, we are going to Loop until VLAST has exceeded some value. Inside the loop we will perform an EIS scan followed by a Galvanostatic charging step. The parameter dialog for the *Loop (Variable)* step is pictured below.



This dialog box lets you select a variable by name. Once you select a variable, different operators become available. These operators are conditional operators. They are:

- lt* - less than. Loop finishes when the variable is less than the value entered.
- le* - less than or equal. Loop finishes when the variable is less than or equal to the value entered.
- gt* - greater than.. Loop finishes when the variable is greater than the value entered.
- ge* - greater than or equal. Loop finishes when the variable is greater than or equal to the value entered.
- eq* - equal. Loop finishes when the variable is equal to the value entered. Should normally be used only to test against an integer.
- ne* - not equal. Loop finished when the variable is not equal to the value entered. Should normally be used only to test against an integer.

The Value is the value against which the variable is compared. It needs to be an appropriate numeric type for the selected variable. For example, an integer cannot be compared to a real number, but rather must be compared to an integer. Pictured below are the entries used for the above example:



This loop can also be combined with user defined variables, which are described in [Using Variables](#).

## Using Variables

Variables are a powerful aspect of the Sequence Wizard. Variables allow you to make dynamic changes to your sequence, based not-only on values you enter, but on values that are measured.

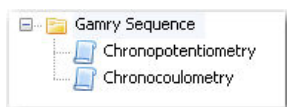
Variables are basically named objects that can be used in place of normal setup parameters. There are 3 variable types, *Potential*, *Real*, and *Integer*. Variables can only be used where the type of the variable is the same as the type of the setup parameter. For example, a Time is usually entered as a real number, so a variable that will be used for a time would be of type *Real*. Cycle numbers are usually integers. Voltages are usually Potentials.

Variables are always given a name. Names are alphanumeric, but must start with a letter from A-Z. There are 2 reserved names, VLAST and ILAST. VLAST is the last measured potential, in volts. ILAST is the last measured current, in amps. At the start of a sequence, these two variables both default to a value of zero. They stay at a value of zero until a step which measures voltage, for VLAST, or current, for ILAST, is run.

## An Example Using VLAST

VLAST is a variable which may commonly be used. For example, you may wish to galvanostatically charge a battery for a specified amount of time. Next you may wish to potentiostatically hold the battery at the last measured voltage and measure the amount of charge that is flowing. This is an instance where VLAST would be used.

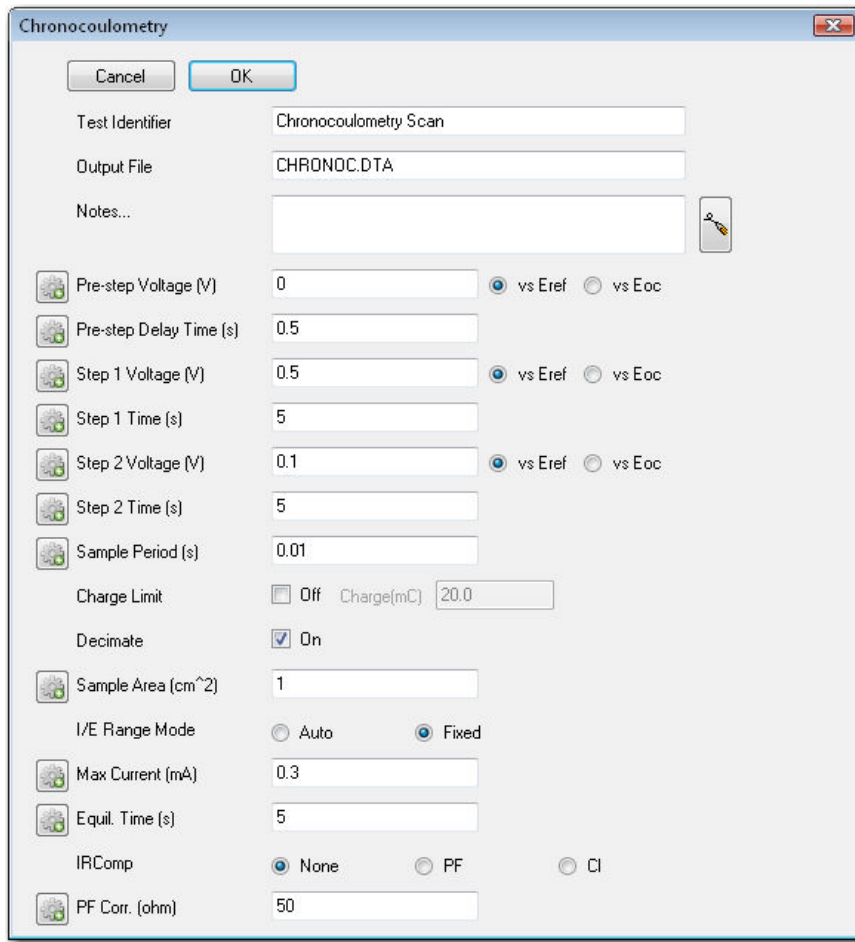
The Sequence would look like:




Notice that there is no mention of a variable in this sequence tree. That is because VLAST is always defined. So let's see how to use VLAST.

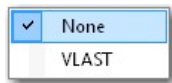
1. Simply bring up the Chronocoulometry setup which looks like the picture below:



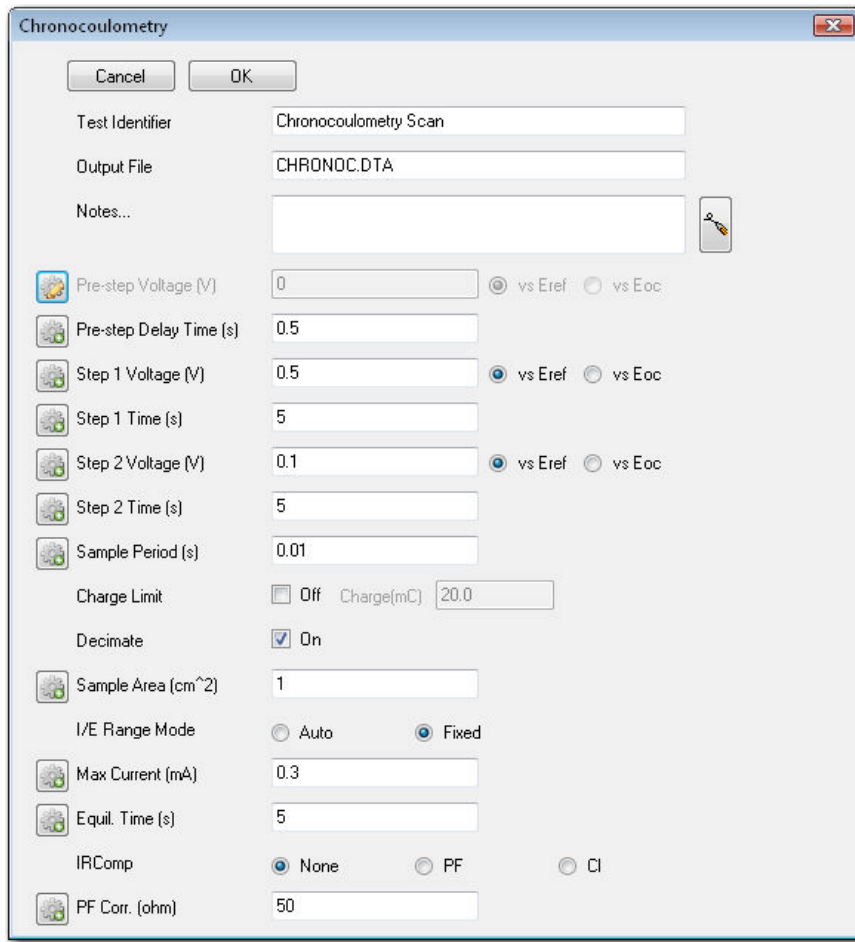



Note that there are 3 different voltages in this step, *Pre-step Voltage*, *Step 1 Voltage*, and *Step 2 Voltage*. For our simple example, we will want to make all of them the same, and make all of them equal to VLAST.

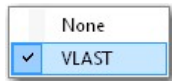
2. Click on the Variable Information Icon  next to the Pre-step Voltage setup parameter. You should end up with a menu that looks like the following:



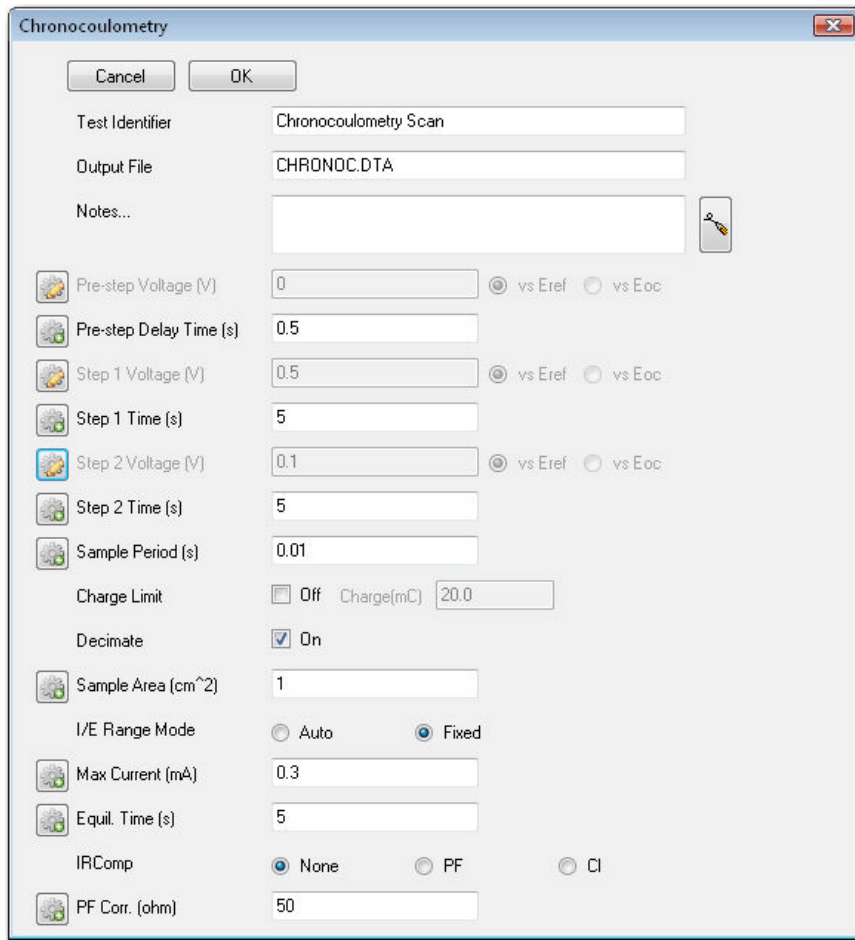
Select VLAST from the menu. The parameter dialog box will now look like:



Notice that the Pre-step Voltage parameter is now grayed out, and that the Variable Information Icon has changed. A Variable Information Icon that looks like  indicates that a variable is being used. If you click on the Variable Information icon now, you should see that VLAST is selected.



3. Repeat this process for the Step 1 Voltage and the Step 2 Voltage. Ultimately the parameter dialog box should look like:

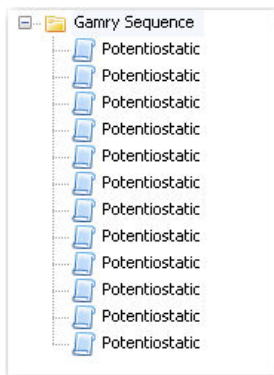


4. The sequence setup is complete. The Chronocoulometry step will now use the last measured voltage from the previous step, Chronopotentiometry.

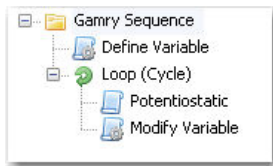
## An Example of Using a User Defined Variable

Now, lets describe an experiment where a user defined variable might be used. Let's say you want to perform a constant voltage experiment (potentiostatic hold) at 12 different, but evenly spaced voltages. Sure, you could create a sequence with 12 different *Potentiostatic* steps, or you could simply use 4 steps, a *Define Variable* step, a *Loop (Cycle)* step, a *Potentiostatic* step, and a *Modify Variable* step.

Lets take a look at what these 2 sequences might look like. First, the 12 step method:



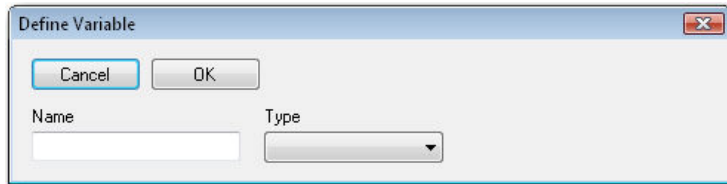
and now the experiment using a variable:



You can see that the variable sequence has less for you to setup. You have less work to do. Rather than editing each of the 12 steps, entering a different voltage and a different filename for each, you can edit only 4 steps.

Let's walk through using the variable.

1. Bring up the parameter dialog box for the *Define Variable* Step. It should look similar to the one pictured below:

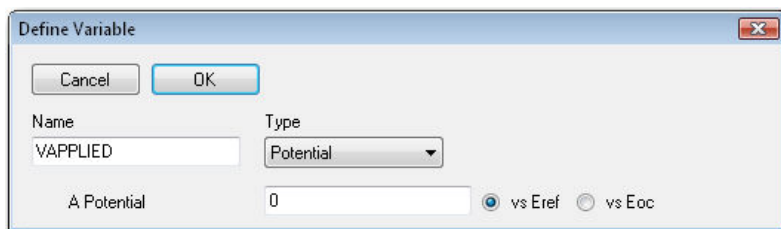


You can see that there are two controls to edit. The first is the **Name**, the second is the **Type**.

**Name** - The name of the variable.

**Type** - The type of the variable.

2. Lets enter our name, VAPPLIED, and select our type. In this case, our type will be a Potential. The parameter box will now look like:



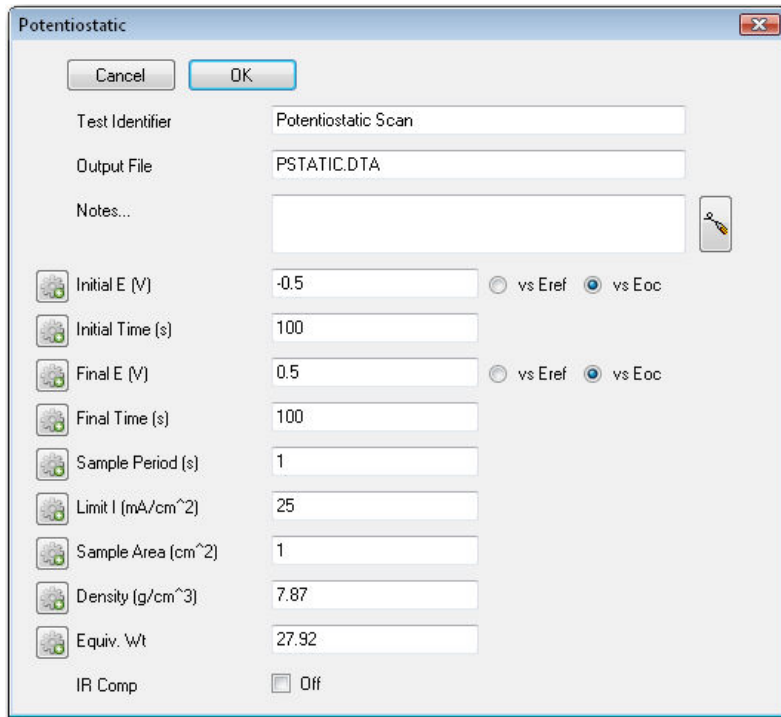
You can see that a new set of controls appeared below the **Name** and **Type** controls. Using these controls is how you specify the initial value of your variable. In our case, we will keep the default of 0 V versus Eref. Press OK to save your variable.

3. Now that we have defined our variable, VAPPLIED, lets set our Loop Count parameter. Bring up the parameter dialog for the *Loop (Cycle)* step by double-clicking on the step, or right clicking on it and selecting **Properties**. Enter a Loop Count of 12. Your parameter dialog should look like the following:




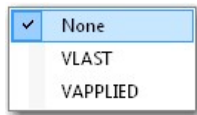
Press OK to save the Loop Count.

4. Now we are ready to use our variable. Bring up the parameter dialog for the Potentiostatic step by double-clicking on the step. Your parameter dialog box should look like:



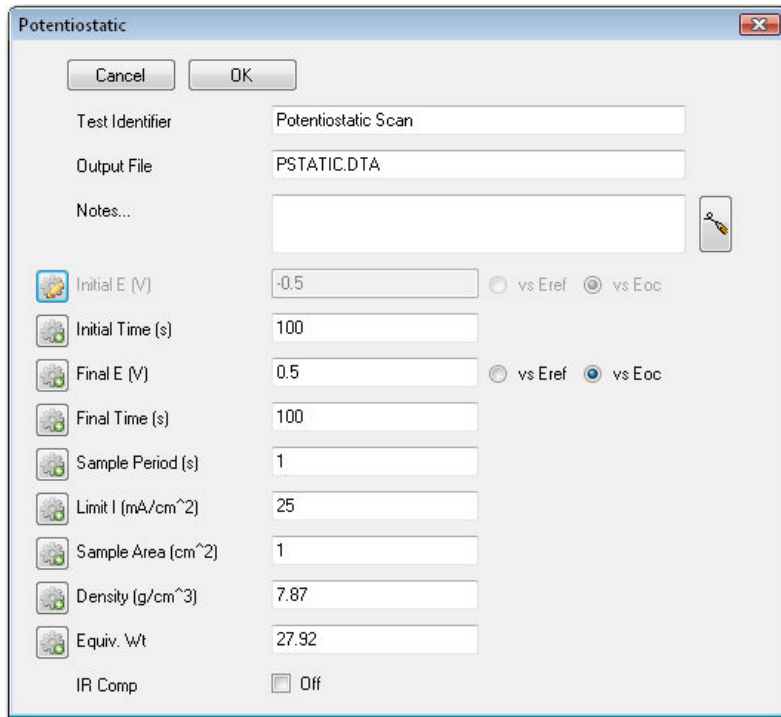
Now, you can see that there are 2 different potentials in this step. *Initial E* and *Final E*. We are going to substitute our variable for these 2 setup parameters.


5. Click on the Variable Information Icon  next to the Initial E setup parameter. You should end up with a menu that looks like the following:

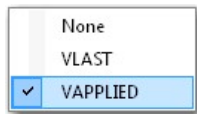


By default, None is selected. But notice that there are 2 variable names also on the list. VLAST and VAPPLIED. We want to select VAPPLIED. Do so by clicking on it.

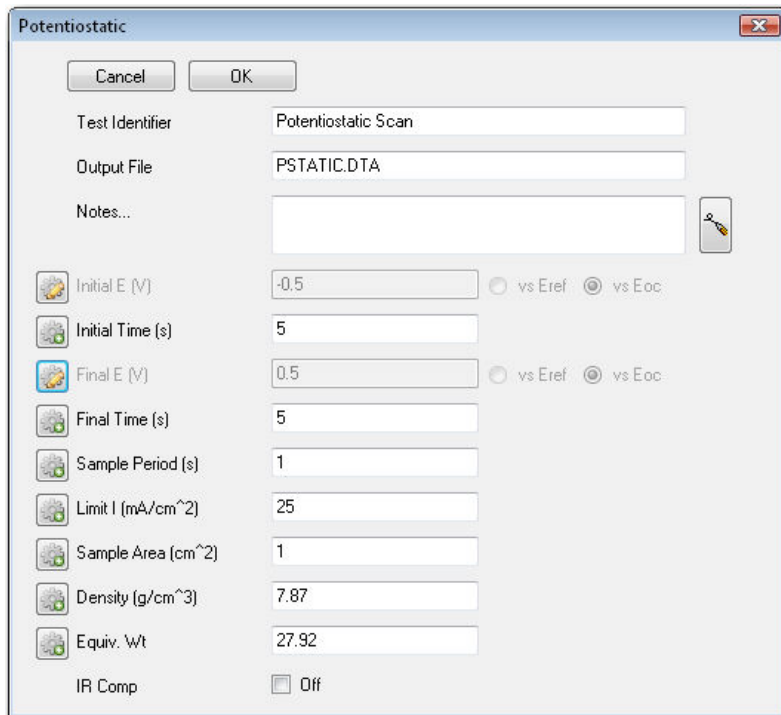
You will then see the parameter dialog look like the following:



Notice that the Initial E parameter is now grayed out, and that the Variable Information Icon has changed. A Variable Information Icon that looks like  indicates that a variable is being used. If you click on the Variable Information icon now, you should see that VAPPLIED is selected.



- Now lets do the same thing for Final E, since we want the same potential applied throughout the step. Also, shorten the times from 100 to 5 seconds for both Initial Time and Final Time. Ultimately the parameter dialog box should look like:



Once you are satisfied with your changes, select OK.

- Now our final edit we need to make is to the Modify Variable step. You can see that this step is inside our loop. It will be run after every *Potentiostatic* step. Bring up its parameter dialog by double-clicking on the step. It should look like:

It starts out empty. You first need to select the variable you wish to modify from the Variable Name drop-down list. Click on this list and you should see a menu like:

Select VAPPLIED from the list. The parameter dialog should now look like:

Notice that the Modifier drop-down list is now enabled. You can select from 1 of 4 entries as shown below:

The selections are:

- + : addition
- : subtraction
- \* : multiplication
- = : assignment

In our case we want to use addition. So select the plus sign from the list. You will then see that the Value edit field becomes enabled. Enter a Value of 0.1. This will increment our VAPPLIED value by 0.1 V every time through the loop. The parameter dialog should end up looking like:

Press OK to save changes.

- Now we have our sequence defined. Our initial voltage is 0 V versus Eref. After the first *Potentiostatic* step, we increment VAPPLIED to 0.1 V. After another *Potentiostatic* step we increment to 0.2 V and so on until all 12 steps have been performed, and our final voltage is 1.1 V. The sequence will then end.