734 Louis Drive, Warminster, PA 18974, USA
+215-682-9330 Fax: +215-682-9331
info@gamry.com www.gamry.com

Programming Reference for Explain™

Explain™ Experimental Control Language

The data acquisition procedures for most electrochemical techniques in Gamry electrochemical instruments 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 Software.

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 Gamry redesigned and simplified it. 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 document gives you the fundamental information necessary to understand what happens in an Explain script. Explain library functions for each Gamry application are described in that application's on-line Help Section.

Data acquisition, of course, is only one part of the experiment. Data analysis is also a key function. Gamry uses the Echem Analyst software platform to perform the analysis portion of the experiment. The Echem Analyst consists of a number of scripts written in Microsoft® Visual Basic for Applications. Like Explain, these VBA scripts can be opened and modified for special applications. Gamry's Open Source Scripting is based on Explain and VBA Scripts.

For a more general discussion of the use of Explain, see the Application Note "Customizing Electrochemical Experiments with Explain™''.

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 are relatively 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, Wordpad, 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.

WARNING: 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 "explain.exp"              ; read in standard Framework library

 global Pstat
global Output
function Main()
        Pstat = PC3.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 lt 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 lt 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 gt 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 Explain script. 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 "explain.exp" which is included in each standard script. You should also include this file in each script you write. Use the line:

```
include "explain.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 <u>incorrect</u> 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 cannot permanently store a value in a local variable. For example, consider the following <u>incorrect</u> 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)
        printl("Modify: Speed = ",Speed)
        Speed = Speed + 1.0
        printl("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, and 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.

BOOL

A BOOL, (short for Boolean) is a true or false value. Explain understands two BOOL constants, TRUE and FALSE. Note that these values must be capitalized and may not be quoted ("TRUE" will be interpreted as a string). Don't confuse the BOOL data type with Boolean operators: and, or, xor.

The allowed primitives operations are:

BOOL Equality BOOL -> BOOL (TRUE eq TRUE -> TRUE.)

BOOL Boolean BOOL -> BOOL (TRUE xor TRUE -> FALSE.)

Unary BOOL -> BOOL (not TRUE -> FALSE)

INDEX

INDEX is a signed 32 bit integer ranging from -2^31 to +2^31-1.

Any numeric constant without a decimal point has data type INDEX (e.g. 2, -15, 1238274). If a number has a decimal point, it will be translated as a REAL (e.g. 2.0 is a REAL) .

The allowed primitive operations are:

INDEX Mult INDEX -> INDEX (-2 * 2 --> -4)

INDEX Sum INDEX -> INDEX (1 + 1 --> 2)

INDEX Shift INDEX -> INDEX (1 << 2 --> 4)

INDEX Equality INDEX -> BOOL (1 eq 1 --> TRUE)

INDEX Comparison INDEX -> BOOL (1 > 2 --> FALSE)

BITS Shift INDEX -> BITS (0x0002 >> 1 -> 0x0001)

BITS

BITS is a 32 bit array of bits, i.e. true and false flags.

A BITS constant is a Hexadecimal value beginning with a '0x' prefix as in the C language (e.g. 0x001A).

The allowed primitive operations are:

BITS Boolean BITS -> BITS (0xF1 and 0x10 -> 0x10)

BITS Shift INDEX -> BITS (0x2 << 2 -> 0x8)

Unary BITS -> BITS (not 0xAAAA -> 0xFFFF5555)

BITS Equality BITS -> BOOL (0xAAAA eq 0xAAAB -> FALSE)

STRING

A STRING is an ASCII character string.

String constants are delineated by double quotes: "This is one string". A string constant has the same special control characters as the C language. Special codes must be preceded with the backslash character,'\'. For example:

"line one\nline two".

The following control characters are understood:

\b	Bell
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\\	Normal backslash
\"	Normal quote (don't end the string)
\0###	Numeric value translated as single character

Although there are not any primitive operations that can be performed on a string, they are used extensively in inserting information in the script output file, e.g.:

```
print("DATE\t", datestamp())
print("ECORR\t", Ecorr)
```

REAL

REAL is a signed 32 bit floating point number. Explain uses the IEEE 4 byte format equivalent to a C float.

REAL constants must contain a decimal point and at least one digit on both sides of the decimal. They also may have an exponent (e.g. 1.0, -1.0, 1.0E-2, 1.0E2, 11.283).

The allowed primitive operations are:

REAL Sum REAL -> REAL (1.0 + 1.0 -> 2.0)

REAL Mult REAL -> REAL (1.0 / 2.0 -> 0.5)

REAL Equality REAL -> BOOL (1.0 eq 2.0 -> FALSE)

REAL Comparison REAL -> BOOL (1.0 gt -1.0 -> TRUE)

NIL

NIL is a special data type indicating an unassigned or default value. The NIL data type has only one possible value, also called NIL. Any variable that is named but has not had a value assigned to it has a data type NIL. The term NIL is also used to indicate a NIL constant, e.g.:

```
A = NIL.
```

Remember that 'A' in this example is assigned both a data type of NIL and the value NIL.

The only primitive operation used with NIL is a comparison:

data Equality NIL -> BOOL (1.0 eq NIL -> FALSE)

Explain's use of NIL for unassigned variables can cause subtle problems. Since Explain is case sensitive, you can easily make an error like:

```
CurrentRange = 0.1
Pstat.SetIERange(Currentrange)
```

The first variable, 'CurrentRange' is an INDEX with value 1 while the second variable, 'Currentrange', is a NIL! Because SetIERange() can take either a NIL, an INDEX, or a REAL parameter, this code fragment does not generate a compiler error. It also does not do what the programmer intended.

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 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 VRAMP
        license Signal   callin "RampSignal"
...                                          ; 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
```

frees 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 Cpiv 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 Cpiv 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.