PL/B Using OLE/ActiveX Architecture

Sunbelt Computer Software

Day 1. Basics of Automation

1.1 Introduction

Automation (also known as OLE Automation or ActiveX Automation) is a method of allowing applications (such as Word and Excel) to expose their functionality to other applications, including applications written in PL/B.

Automation is based upon the Component Object Model (COM) technology. Automation uses the standard COM interface IDispatch to access the object's Automation interface. Any object that implements IDispatch implements Automation.

The goal was to provide a simple way to access an application's exposed properties and methods in a language independent way. Properties refer to a defined set of data. Methods refer to a group of functions or procedures. A collection of methods, and properties are known as an Automation object.

One application can contain many different types of Automation objects. The Excel application contains an application object, a books object, a book object, a sheets object, a sheet object, a range object, and many more. These objects are arrange in a hierarchy where the application object is at the top, and contains a property that returns a books object. The books object can return a selected book object. The book object can return a sheets object, and a sheets object can be used to create a sheet object.

The hierarchy of objects allows a complex application to expose its' functionality in the same manner a user would use the application.

Automation is not just limited to applications, but is now used as a standard method of connecting components written in different languages. It is also the basis for ActiveX controls, and OLE container support.

1.2 Useful Definitions

ActiveX Control	- A specific type of COM object which supports the **IUnknown** interface and is self-registering. An additional PL/B language requirement is that the control must support at least the **IOleObject** interface and the **IOleControl** interface.
Automation	- A means of allowing applications (such as WORD and EXCEL) to expose their functionality to other applications.
Automation Object	- A collection of methods and properties.
Class	- A class can be an object, or a grouping of similar objects. In many languages, the class is the basic definition of an object, and any specific object is know as an instance of the objects class.
CLSID	- A 128 bit unique identifier for an Automation object.
COM	- Component Object Model technology. COM provides a standard Definition of how objects (such as PL/B runtime, Visual Basic, and Microsoft WORD) can communicate and share data.
IDispatch	- Standard COM interface that exposes an application's internal functions to other software.
Interface	- A declared set of specifications including methods and properties which identifies the manner in which objects can communicate.
IOleControl	- Commonly supported COM interface for ActiveX controls. This Interface allows a container to tell the control that an ambient property has changed or that the container will ignore its events for a while.
IOleObject	- A complex COM interface which lets a client object ask a server object to perform many different actions.
IUnknown	- Fundamental interface required for every COM object. IUnknown contains three methods: QueryInterface, AddRef, and Release.
Methods	- A reference to a group of functions or procedures.
Object	- Something consisting of two elements: a defined set of data (properties) and a group of methods.

OLE	- Object Linking and Embedding. Technology used to create compound documents.
ProgID	- A text name for a CLSID.
Properties	- A reference to a defined set of data.
Registry	- A unified database for storing system and application configuration data in a hierarchical form.

1.3 Component Object Model Architecture

The Component Object Model (COM) provides a standard definition of how objects (such as the PL/B runtime, Visual Basic, or Microsoft Word) can communicate and share data. Objects can communicate through either standard or custom COM interfaces. COM is:

- Operating System independent
- Language independent
- A binary standard for component interoperability
- Able to create and use objects on both local and remote computers

The role of COM is to be the glue between components, allowing interaction between software objects.

To meet the goals stated above, the COM specification (located at www.microsoft.com/com) details the binary interface for component interaction, and connection.

COM supports interaction between in-process objects (objects that reside in the same process memory space), local out-of-process objects (objects that reside on the same computer), and remote out-of-process objects (objects that reside on different computers). For the developer, all objects appear as if they were in-process objects.

The developer sees COM as a small API set, and a series of interfaces. An interface is a strongly typed contract between software components. This contract defines:

- A unique identifier for each interface
- The location of functions within the interface
- The purpose of each function
- The arguments and return values of each function

The interface doesn't define:

- Operating System the object runs on
- Language the object was written in
- Method of function implementation

1.4 Globally Unique Identifiers

Each interface requires a unique identifier. COM solves this problem by using globally unique identifiers (GUIDs) which, as 128-bit integers, are guaranteed to be unique.

A GUID is made up of a time stamp, and a 6 byte MAC address from a network card. A GUID can also be known as a UUID (universally unique identifiers), and can be created by the UUIDGEN.EXE command.

When a GUID is used to identify a component it is known as a class identifier (CLSID), and when used to identify an interface it is known as a interface identifier (IID).

Under Window, each component must register its CLSID under the HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID key. The HKEY_CLASSES_ROOT key can be used as a short form for the HKEY_LOCAL_MACHINE\SOFTWARE\Classes key.

To make CLSIDs more 'user friendly', each CLSID can have associated programmatic identifiers (PROGIDs), which is a readable text string. These identifiers have no more than 39 characters, can not start with a digit, and can use no punctuation other than periods.

Each PROGID – CLSID combination is stored in the registry under HKEY_LOCAL_MACHINE\SOFTWARE\Classes\<PROGID>\<CLSID>

1.5 COM Interfaces

Every COM object must have at least one interface known as IUnknown. Every other COM interface must be derived from IUknown. This interface manages all other interfaces on a COM object. It does this by providing three methods:

- QueryInterface

  Returns a code pointer to a interface, specified by an IID, on an object to which a client currently holds an interface pointer. This function calls IUnknown::AddRef on the pointer it returns.

- AddRef

  This method increments the reference count for a calling interface on an object. It should be called for every new copy of a pointer to an interface on a given object. It returns the value of the new reference count.

- Release

  This method decrements the reference count for the calling interface on a object. If the reference count on the object falls to 0, the object is freed from memory. It returns the value of the new reference count.

Once a COM interface is put into production, the interface is frozen. The COM standard requires that a new interface must be created if functionality is changed.

The Interface Definition Language (IDL) is used to describe COM interfaces. This information is stored in a type file, which can be located in a DLL (.dll or .ocx), Executable file (.exe), or a stand-alone file (.tlb, .olb).

1.6 OLE/COM Object Viewer

The OLE/COM Object Viewer is a developer administration and testing tool. With the OLE/COM Object Viewer you can:

- Browse all of the Component Object Model (COM) classes installed on your machine.
- See the registry entries for each class.
- Configure any COM class on your system. This includes Distributed COM activation and security settings.
- Configure system wide COM settings, including enabling or disabling DCOM.
- Test any COM class, simply by double-clicking its name. The list of interfaces that class supports will be displayed. Double-clicking an interface entry allows you to invoke a viewer that will "exercise" that interface.
- Activate COM classes locally or remotely. This can be used for testing DCOM setups.
- View type library contents. Use this to examine what methods, properties, and events an ActiveX Control supports.
- Copy a properly formatted OBJECT tag to the clipboard for inserting into an HTML document.

The OLE/COM Object Viewer requires either Windows 95, Windows 98 or Windows NT version 4.0 with service pack 3 or later. You must have also installed Internet Explorer 3.0 or later, and DCOM95 if required

The OLE/COM Object Viewer can be obtained from www.microsoft.com/com/resource/oleview.asp

When using the OLE/COM Object Viewer, it is useful to know the meaning of the following IDL tags:

❖ coclass	- A COM object class
❖ [default]	- Default interface for a coclass
❖ [default,source]	- Default events for a coclass
❖ dispinterface	- Indicates a IDispatch interface
❖ typedef enum	- A collection of named constant values

1.7 Automation Object

The AUTOMATION object is implemented as an ActiveX Automation Client interface, which allows PL/B to programmatically control any object/program that conforms to the Automation interface.

Data Declaration Format:

```
<label>  AUTOMATION   [CLASS={slit}]
```

PLB Verb Usage:

```
CREATE    {object}[,{properties}]
DESTROY   {object}
GETPROP   {object}[,{properties}]
SETPROP   {object}[,{properties}]
EVENTREG  {object},{event}[,{routine}[,{keyword=dest}]]
EVENTINFO {pos}[,{keyword=dest}]
LISTINS   {collection},{object}...
LISTDEL   {collection},{object}...
{object}.{method} GIVING {result} USING [{param}][,..]
```

1.8 Automation Object Creation

To create an Automation object in PL/B, the programmer must first code an AUTOMATION data statement, as follows:

```
ExcelApp        AUTOMATION
```

This statement can optionally contain a class identifier, as follows:

```
ExcelApp        AUTOMATION        Class=”Excel.Application”
```

This marks the Automation object as always being a certain type, and will be used in the future for compiler syntax checking.

The next step is to actually create the Automation object using the CREATE statement, as follows:

```
                CREATE        ExcelApp, Class=”Excel.Application”
```

If the class identifier was already specified on the data declaration, it is not required on the CREATE statement, and will cause an error if found.

The class identifier can be either a CLSID or a ProgID(text name for a CLSID). A CLSID is a 128 bit unique identifier for the Automation object.

The information relating ProgIDs to CLSIDs, and CLSIDs to Automation objects is kept in the registry under the HKEY_CLASSES_ROOT key.

When the CREATE statement completes, the Automation object has been created and is ready for access.

When you are finished using the Automation object, it can be destroyed by using the DESTROY statement.

1.9 Automation Object Properties

The properties of an Automation object can be accessed through extensions made to the GETPROP and SETPROP statements. These statements are as follows:

```
GETPROP   {object}[,{properties}]
SETPROP   {object}[,{properties}]
```

Where:

{object} - AUTOMATION label reference.

{properties} - The properties for an AUTOMATION object are available as two groups. The first group is defined by the normal PL/B property set. This first group of properties is under direct control of the runtime. The second group of properties is provided only via the Automation interface defined for the CLASS identifier. The user must use vendor supplied documentation to define the specific Automation interface properties available.

Group One (Direct/Static Properties)

CLASS={svarslit}
CLASSID={svarslit}
LIC={svarslit}
OBJECTID={dnumnvar}

Group Two (IDispatch Properties)

The properties available through the IDispatch interface specified use the following syntax:

*{name}={value}
@{svar}={value}

Where:

{name} - IDispatch property name. Property names are not case sensitive.

{svar} - Specifies a DIM variable which contains the IDispatch parameter name.

{value} - Values can be a svar, nvar, VARIANT object, or AUTOMATION object.

There is one special property, $IDispatch, that can be used to get the IDispatch interface from a collection. The receiving variable must be either an Automation object or a Variant object. This will be covered in more detail in the section on object collections. The simplest form of getting a property is as follows:

```
IsVis   Integer       4
        GetProp       appExcel,*Visible=IsVis
```

Setting a property can be just as simple:

```
        SetProp       appExcel,*Visible=1
```

These statements could also have been written as follows:

```
VisName       Init          “Visible”
IsVis         Integer       4

        GetProp       appExcel,@VisName=IsVis
        SetProp       appExecl,@VisName=1
```

By using the @ indicator generic Setprop and Getprop statements can be written that be modified during runtime.

1.10 Automation Object Methods

The methods of an Automation object can be accessed through a new method statement type. This new statement type is as follows:

```
{object}.{method} GIVING {result} USING [{param}][,..]
```

Where:

{object}	- AUTOMATION label reference.
{method}	- When an object method is being used, the {method} name identifier is determined by referencing the vendor documentation. The {method} names are not case sensitive. The {method} can be specified either as a keyword form or as a {svar} variable with a leading '@' character. The acceptable syntax forms are as follows: {object}.methodname {object}.@{svar}
{result}	- The {result} is the expected return value for the {method} being executed for the specified {object}. The variable types for {result} can be {svar}, {nvar}, VARIANT object, or AUTOMATION object.

{param}	- The object method parameters identify the expected values required for the {method} of a specified {object}. The keyword name of a given {param} can be determined from the Automation vendor documentation. The {param} keyword names may or may not be used. If the {param} name is not specified, then the parameter must be placed into the {param} list at a valid position as documented for the method. The {param} can have one of the following syntax forms:
	
	{value}
	*{name}={value}
	@{svar}={value}
	
	Where:

{name}	- IDispatch property name. Parameter names are not case sensitive.
{svar}	- Specifies a DIM variable which contains the IDispatch property name.
{value}	- Values can be a svar, nvar, VARIANT object, or AUTOMATION object.

This new form of statement allows each Automation object used to extend the PL/B language.

The simplest Automation statement is one where no arguments are passed, and no return value is given, as shown below:

```
        appExcel.Quit
```

The same method of generic coding, used with properties, can also be used with methods as shown below:

```
QName       Init    “Quit”
            AppExcel.@QName
```

When expecting a return value, the GIVING keyword must be used as shown below:

```
SheetsObj   Automation
Sheet       Automation

        SheetsObj.Item      Giving Sheet Using 1
```

In this sample the Item method is used to return an Automation object named Sheet, using an index of 1.

Arguments are placed after the USING keyword, and can be optionally named. The following four code lines are the same:

```
        Charts.Add    Using  100, 10, 350, 250
        Charts.Add    Using *Left=100, *Top=10, *Width=350, *Height=250
        Charts.Add    Using *Left=100, *Width=350, *Top=10, *Height=250
        Charts.Add    Using  100, *Width=350, *Top=10, *Height=250
```

Remember that once a named argument is used, all arguments after must also be named.

Named arguments can also be coded in a generic form as shown below:

```
TopName     Init    “Top”

        Charts.Add    Using 100, @TopName=10, *Width=350, *Height=250
```

1.11 Automation Object Collections

Sometimes an Automation object will be used to control other Automation objects. This Automation object is known as an Automation collection object, and must use a standard interface. A collection has two standard methods, two optional methods and one property as described below:

- _NewEnum

  This is a required method. This method will return a pointer to an IEnumxxxx interface.

- Item

  This is a required method. This method will return an Automation object from the collection. The arguments passed in are used to identify the Automation object requested, and are implementation dependent. In many cases this will be either a numeric index, or an identifying name.

- Add

  This method is optional, and only needed if Automation objects can be added or removed from the collection. The arguments passed in are implementation dependent, and the result is the newly created Automation object.

- Remove

  This method is optional, and only needed if Automation objects can be added or removed from the collection. The arguments passed in are used to identify the Automation object to be removed, and are implementation dependent. In many cases this will be either a numeric index, or an identifying name.

- Count

  This is a required property and returns the number of objects in the collection.

Collections are found as ‘get’ only properties. An example collection would be a collection of spreadsheets found in an Excel workbook. The following code would get an Automation object pointing to a specific spreadsheet using an index to the first sheet in the collection:

```
GetProp          WrkBook,*Sheets=SheetsObj
SheetsObj.Item   Giving Sheet Using 1
```

For simple cases this syntax is acceptable, but take the following code to store a value in a cell:

```
GetProp           Sheet,*Range=RangesObj
RangesObj.Item    Giving Range Using “A6”, “A6”
SetProp           Range,*Value=”Orange”
Destroy           Range
Destory           RangesObj
```

To reduce the coding effort an extended syntax has been created for methods, GETPROP, and SETPROP statements. This extended syntax allows the collection information to be placed within the context of the object variable. The new syntax changes the lines of code above to:

```
SetProp      Sheet.Range(“A6”,”A6”),*Value=”Orange”
```

The PL/B runtime will create and destroy any temporary Automation objects needed to get to the final destination.

The extended syntax allows the object variable component to contain up to ten collection references, A collection reference consists of the name of the property that holds the collection, and the indexes for the Item method enclosed in parentheses. The following code will clear out a speadsheet:

```
Sheet.Range(“A6”,”W40”).Clear
```

This statement will invoke the Clear method obtained from the Item method of the Automation collection returned by the Range property of the Sheet Automation object. The indexes used would be the strings “A6” and “W40”.

Points to remember are that methods must end with the name of an actual method, and properties must not end with a method.

In some cases a Automation object must be passed to another routine. To help with this situation a special property named $IDispatch has been added. This property will return the last temporary Automation object created by the run-time. The following code obtains an Automation object to a spreadsheet range:

```
Range         Automation

        GetProp      Sheet.Range(“A6”,”W40”),*$IDispatch=Range
        Range.Clear
```

All special properties and methods will start with a $ symbol.

1.12 Variant Objects

One problem faced in component software is the transfer of data. All languages have defined their own specific data types, and have no common set to draw on. Automation has solved this problem by creating a standard set of data elements to be used between the Automation controller and Automation server. The structure that defines these data elements is known as a VARIANT.

To allow PL/B programs full access to the Automation world, a new PL/B object that supports this data structure was created.

Data Declaration Format:

```
<label>  VARIANT
```

PLB Verb Usage:

```
CREATE    {Variant object}[,{properties}]
DESTROY   {Variant object}
GETPROP   {Variant object}[,{properties}]
SETPROP   {Variant object}[,{properties}]
LISTINS   {collection},{Variant object}...
LISTDEL   {collection},{Variant object}...
```

Properties:

VARVALUE={svar|nvar|ivar|AUTOMATION object|VARIANT object}

The VARVALUE property allows the value of the VARIANT to be assigned in accordance with the VARTYPE of the object.

VARTYPE={dnumnvar}

The VARTYPE property identifies the VARIANT data type as allowed for conversion between PLB data types and applicable Windows variant types. The following table identifies valid VARTYPE {dnumnvar} type values:

$VT_EMPTY	EQU 0	nothing
$VT_NULL	EQU 1	SQL style Null
$VT_I2	EQU 2	2 byte signed int
$VT_I4	EQU 3	4 byte signed int
$VT_R4	EQU 4	4 byte real
$VT_R8	EQU 5	8 byte real
$VT_CY	EQU 6	currency
$VT_DATE	EQU 7	date
$VT_BSTR	EQU 8	OLE Automation string

$VT_DISPATCH	EQU 9	IDispatch FAR*
$VT_ERROR	EQU 10	SCODE
$VT_BOOL	EQU 11	True=-1, False=0
$VT_VARIANT	EQU 12	VARIANT FAR*
$VT_UNKNOWN	EQU 13	IUnknown FAR*
$VT_UI1	EQU 17	unsigned char

VARPTR={dnumnvar}

The VARPTR property identifies when a VARIANT is to be passed to an OLE application for a property/method parameter as a VT_VARIANT. This property causes the runtime to pass a VT_VARIANT to the interface where the VT_VARIANT contains the address of the variant data itself. The requirement for passing a VT_VARIANT is controlled by the OLE application being used by the PLB program. The {dnumnvar} can be set to zero ($FALSE) which indicates the VARIANT object is to be processed as a VARIANT, which is the default. If the {dnumnvar} value is set to a non-zero value ($TRUE), then this indicates that the VARIANT object is to be passed to the OLE interface as a VT_VARIANT.

When the VARTYPE of an object is changed, the value of that object is automatically converted to the new type.

Variant objects can also be used in special cases as indicators. When a program wishes to indicate that an optional parameter is not being used, the following type of variant can be created:

```
OptVar                      Variant
VT_ERROR                    EQU         10
DISP_E_PARAMNOTFOUND        Integer     4,"0x80020004"
.
. Lets make up a standard optional variant
.
        Create      OptVar,VarType=VT_ERROR:
                    VarValue=DISP_E_PARAMNOTFOUND
```

The following table gives a breakdown of some possible interface variant types.

* [V] - may appear in a VARIANT
* [S] - may appear in a Safe Array

Microsoft Reference	Possible Usage	Microsoft Description	PLB Cross Reference
VT_EMPTY	[V]	nothing	NA
VT_NULL	[V]	SQL style Null	{svar} NULL
VT_I2	[V][S]	2 byte signed int value limited -32767 to 32767.	{nvar|nlit|ivar}
VT_I4	[V][S]	4 byte signed int value limited to -2147483647 to 2147483647.	{nvar|nlit|ivar}
VT_R4	[V][S]	4 byte real	{nvar|nlit|ivar}
VT_R8	[V][S]	8 byte real	{nvar|nlit|ivar}
VT_CY	[V][S]	currency limited to 2 decimal places.	{nvar|nlit}
VT_DATE	[V][S]	date format determined by usage.	{svarslit}
VT_BSTR	[V][S]	OLE Automation string	{svar|slit}
VT_DISPATCH	[V][S]	IDispatch FAR*	AUTOMATION object
VT_ERROR	[V][S]	SCODE	{nvar|nlit|ivar}
VT_BOOL	[V][S]	True=-1, False=0	{nvar|nlit|ivar}
VT_VARIANT	[V][S]	VARIANT FAR*	VARIANT object with VARPTR property set.
VT_UNKNOWN	[V][S]	IUnknown FAR*	NA
VT_UI1	[V][S]	unsigned char value limited 0 to 255.	{nvar|nlit|ivar}

1.13 IDispatch Interface

Every Automation object that can be accessed through PL/B must have an interface known as IDispatch. This interface allows PL/B to perform late-binding on the methods and properties specified in a PL/B program. It does this by providing four methods:

- GetIDsOfNames

  Translates a member name and an optional set of argument names to a series of dispatch identifiers (DISPIDs). These DISPIDs are then used when calling the IDispatch::Invoke function.

- GetTypeInfo

  Returns an interface pointer to the type library information.

- GetTypeInfoCount

  Returns the number of type interfaces supported by the object. The value will be zero if no interface is supported or 1 if a type interface is supported.

- Invoke

  Provides a means of accessing the objects properties and methods.

When an Automation object is created, PL/B obtains an interface pointer to the IDispatch interface. To execute a method, the run-time will use IDispatch::GetIDsOfNames to obtain the DISPID for the method and any named arguments. It will then use these DISPIDs to call the IDispatch::Invoke function.

1.14 Automation Objects and the Registry

Automation objects are listed in the registry under HKEY_CLASSES_ROOT by ProgID. The ProgID entry contains a sub-key with the value of the CLSID.

This CLSID value is used to look up CLSID under HKEY_CLASSES_ROOT/CLSID. The following standard keys can be found under the CLSID entry:

❖ InprocServer32	- Name of .dll for the automation object.
❖ LocalServer32	- Name of the .exe for the automation object.
❖ ProgID	- Program Identifier for this CLSID.
❖ Typelib	- CLSID of the type library information.
❖ Programmable	- Object can be used for automation.
❖ Insertable	- Object can be used in an OLE container.
❖ Control	- Object is an ActiveX control
❖ Implemented Categories	– A series of CLSIDs indicating what this object provides. This can replace the Control, Insertable and Programmable keys.

The Typelib CLSID value is used to look up the type library information under HKEY_CLASSES_ROOT/TYPELIB. The following standard keys can be found under this entry:

❖ <langid>/win32	- Name and location of the type library.
❖ Flags	- Type library flags.
❖ Helpdir	- Location of the help files.

Automation objects are self-registering. A command can be executed using the /Register command line parameter. A .dll or .ocx file can be registered or unregistered using the regsvr32.exe program.

1.15 Visual Basic Primer

Most documentation provided for Automation objects and ActiveX controls has been written for the Visual Basic (VB) programmer. This section will provide a quick PL/B to VB guide.

Creating an automation object in VB can be done using the CreateObject function as shown below:

```
Set appAccess = CreateObject("Access.Application")
```

Or

```
Dim appAcc As Access.Application

        Set appAcc = New Access.Application
```

In PL/B:

```
AppAccess           Automation

        Create  AppAccess,Class="Access.Application"
```

To destroy an automation object in VB, it is set to Nothing.

```
Set appAccess = Nothing
```

In PL/B

```
Destroy appAccess
```

A property can get obtained or set in VB by simply assigned the property to a variable or assigning a value to a property, as shown below:

```
        flag = appExcel.Visible
        appExcel.Visible = TRUE
```

In PL/B

```
        GetProp             appExcel,*Visible=flag
        SetProp             appExcel,*Visible=True
```

A method in VB is invoked in a similar fashion to PL/B. The arguments are enclosed with parenthesis, and a named argument is in the form <name>:=<arg>. The syntax for collection access is almost identical, as shown below:

```
appExcel.Workbooks.Open("C:\Sample.xls")
Set maiMail = appOutl.CreateItem(olMailItem)
```

In PL/B

```
appExcel.Workbooks.Open         Using "C:\Sample.xls"
appOutl.CreateItem                      Giving maiMail Using olMailItem
```

VB programmers also have the use of two other special statements. The With statement and the For Each statement.

The With statement allows you to set multiple properties or methods on the same object within one code section. The following example shows the use of this statement:

```
With appAcc
        .OpenCurrentDatabase "nwind.mdb"
        .DoCmd.OpenForm "Employees", acNormal
        .Visible = True
        .CloseCurrentDatabase
        .Quit
End With
```

In PL/B

```
        appAcc.OpenCurrentDatabase      Using "nwind.mdb"
        appAcc.DoCmd.OpenForm   Using "Employees", acNormal
        SetProp                 appAcc,*Visible = True
        appAcc.CloseCurrentDatabase
        appAcc.Quit
```

The For Each statement is used to take the same action or set the same property for all of the objects in a collection. The following example has no simple equivalent PL/B code:

```
For Each Document In appWord.Documents
        Document.Visible = True
Next
```

In PL/B

```
        GetProp         AppWord.Documents,*Count=count
        Move            "0" To Pos
        Loop
          SetProp       AppWord.Documents(Pos),*Visible=TRUE
```

```
 Add         “1” To Pos
Repeat       While (Pos < count)
```

Day 2. OLE Container Support

2.1 Introduction

An OLE (Object Linking and Embedding) container object is an object that can contain embedded or linked data from any application that can be an OLE server. The job of the OLE server application is to present and manipulate the data for the OLE client application.

OLE container object support is based upon the Component Object Model (COM) technology. It also uses Automation, structured storage, in-place activation, and uniform data transfer technologies.

An OLE container object can either contain data or be empty. This state can change back and forth may times during the lifetime of an OLE container object.

This object can be used for such things as displaying an Excel chart, showing a movie, allowing a user to enter data into a Word document, or hosting an ActiveX control. From the user's perspective, it is the PL/B program that is performing all these actions.

The PL/B runtime requires that every object that is to be placed into an OLE Container supports both the IOleObject interface and the IPersistStorage interface. In addition the PL/B runtime will use the IViewObject, IViewObject2, IOleLink, IDataObject, IDispatch, IOleInPlaceObject, and IOleInPlaceActiveObject interfaces.

A PL/B form will provide support for the IOleInPlaceFrame, IOleItemContainer, IDropTarget, and IDropSource interfaces.

The OLE container object provides support for the IOleClientSite, IAdviseSink2, and IDispatch interfaces.

Since OLE container support is built on top of Automation, any OLE container object can be treated as an Automation object.

2.2 Compound Document Files

The original use for OLE was to produce compound document files. The technology would allow a user to create a document containing many different types of data, without the application creating the document knowing how to manipulate all the data types. This allowed a word processing application to include spreadsheets, pictures, and movies into a word processing document, by using other applications to control the spreadsheets, pictures, and movies.

The first problem to solve was the actual storage of data that the application had no knowledge of how to store. To solve this problem, a new type of file, named a compound file, was created. A compound file is like a file system unto itself. It contains two types of objects:

- Storage Object

  This object is similar to a directory in a file system. Storage objects can contain stream objects, and other storage objects. Every compound file must have a root storage object. Each storage object can be marked with the CLSID of it's creator.

- Stream Object

  This object is similar to a file in a file system. It contains either the actual data for an embedded object, or information on the location of a file for a linked object. Each stream object can be marked with the CLSID of it's creator.

The controlling application can give each OLE server application it's own storage object to save its data. When the document is re-loaded, the storage object is passed back to OLE server application for manipulation.

This method of structured storage also allows for incremental access, multiple use, transactional processing on a storage level, and saving of documents in low memory conditions.

Each PL/B form can be a compound file, containing the initial data for OLE container, or an ActiveX control. When the PL/B runtime is started, a temporary compound file is created to support OLE container objects, and ActiveX controls. If an OLE container object is created from a form, and contains some initial data, that data will be transferred to the temporary compound file. If the data is changed, it is changed only in the temporary compound file, and is not saved back to the form.

2.3 In-Place Activation

The second feature needed for the acceptance of the OLE technology was the ability to seamlessly integrate the interaction between the OLE container application and the OLE server application, so that the user would not know that two different applications were in use. To solve this problem, in-place activation was introduced. This allowed the OLE container application at act as if the manipulation of an OLE object was being performed by itself.

The in-place activation or visual editing of an OLE object causes the OLE container application's interface to change to incorporate the features of the component application that created the embedded item. This could mean the replacement of menus, and the addition of toolbars. It also gives keyboard focus and a visual indication that the OLE object is being edited.

In-place activation is not just limited to editing. For example a movie or a sound might start to play.

It is also possible to edit the embedded data using a separate window created by the OLE server application. When this occurs, the OLE object being edited will be displayed as a shaded object, to indicate that it is being edited in an outside window.

Linked OLE objects are never in-place activated because the actual data for the item is contained in a separate file, out of the context of the application containing the link. Editing a linked item in a separate window reminds the user that the data belongs to a separate document.

Only one OLE container object on each form can be in-place activated at one time.

2.4 OLE Container Object Life Cycle

An OLE container object's initial state is to be empty. This means that it contains no OLE insertable object.

During the CREATE statement, or by using the CLASSID, or SOURCEDOC properties, the OLE container object can go to the loaded state. This indicates an OLE insertable object has been loaded into the OLE container object.

If the CLASSID property is used, a new OLE insertable object will be created by the OLE server application specified by the value of the property.

If the SOURCEDOC property is used, either a linked or embedded OLE insertable object will be created using the data from the filename specified by the value of the property. A linked OLE insertable object will be created if the OLETYPEA property is set to $OLETYPELINK. Otherwise an embedded OLE insertable object will be created.

Once the OLE container object is in the loaded state, the OLE container object will either display an icon representing the data, or display the contents of the data. The DISPLAYTYPE property is used to control the type for display. If the content is displayed, the SIZEMODE property can be used to clip, scale, or stretch the visual representation within the area of the OLE container object.

If an OLE server application is not available to manipulate the OLE insertable object, the OLE container object will always remain in the loaded state. This means that although the data is visible, it can't be modified.

If an OLE server application is available, then the OLE insertable object can be place in one of four different states. These states are running, active in a separate window, in-place active, and user interface (UI) active.

The first state after the loaded state is the running state. This takes place when the OLE server application is loaded into memory and is executed. The PL/B runtime will automatically perform this state change when needed.

From the running state, the OLE insertable object can go to either the active in a separate window state, or the in-place active state. Once in the in-place active state, the transition to the UI active state can be performed.

These state changes are performed by the means of using the OLE verb interface. Each OLE insertable object can be instructed to perform actions based upon a set of exported action verbs. This set of verbs is dependent on the type of the OLE insertable object, but a default set is also provided.

Interaction with the OLE insertable object can be performed through the AUTOACT property, the AUTOVERB property and the use of the $DoVerb method.

If the AUTOVERB property is set to be true, the OLE container object will support a context menu containing all the verbs for the OLE insertable object. Performing a right mouse button click over the OLE container object can activate this menu.

The AUTOACT property is used to cause the runtime to execute the OLEPrimary verb under certain circumstances. The OLEPrimary verb causes the OLE insertable object to perform it's default action.

If the property is set to $AUTOACTDBLCLICK, the OLEPrimary verb will be executed when a double-click occurs on the OLE container object. It will also occur when the OLE container object has the focus and the enter key is pressed.

If the property is set to $AUTOACTGETFOCUS, the OLEPrimary verb will be executed when the OLE container object gets the focus.

If the property is set to $AUTOACTMANUAL, the PL/B runtime will not automatically execute any verb. The programmer must use the $DoVerb method instead.

The $DoVerb method allows the programmer to directly control the OLE insertable object. By using the OLEOpen verb, the programmer can cause the state to change to the active in a separate window state. The OLEHide verb is used to transition back to the running state.

When the OLE insertable object is in the active in a separate window state, the HOSTNAME property can be used to allow the OLE server application to indicate what program it is linked to.

The OLEInPlaceActivate can be used to place the OLE insertable object into the in-place active state, allowing the OLE insertable object to be in-place edited.

If merged menus and toolbars are wished, the OLEUIActivate can be used to transition from the in-place active state to the UI active state. When in the UI active state, the program can allow or prevent menu and toolbar integration by using the NEGMENU (negotiate menu) and NEGBORDERS (negotiate tool space borders) properties. The <l, <m, and <r indicators have been added to the menu name string to indicate if the menu should be melded to the left (<l), melded in the middle (<m) or melded on the right (<r) when the OLE insertable object and PL/B form menus are integrated.

When the OLE container application loses focus, it transitions back to the running state.

2.5 OLE Drag and Drop

One other technology associated with OLE is the ability to drag an OLE insertable object from one application and drop this object onto another application. This technology was developed as a faster means of the cut/copy/paste paradigm.

OLE drag and drop is based on the Uniform Data Transfer (UDT), IDropTarget, and IDropSource interfaces.

To perform an OLE drag and drop operation, the user first uses the mouse to select the object to be dragged. This is done by clicking and holding the left mouse button down over the object. If the control key is also held down, the object is copied, otherwise the object is just moved.

At this point the mouse cursor is changed to indicate an OLE drag and drop operation is in progress.

The user then drags the mouse cursor over to the area that is to receive the OLE insertable object. When the mouse button is released, the operation is complete.

If the receiving side rejects the operation, no action takes place. Otherwise the OLE insertable object is either copied or moved.

When an OLE insertable object is in-place active, the OLE insertable object provides the support for both OLE drag and drop.

If an OLE container object is empty, the OLEDROP property can be used to indicate if the OLE container object will accept an OLE drag and drop operation. The type of data that can be dropped is controlled by the OLETYPEA property.

2.6 OLE Container Object

The CONTAINER object is implemented as an OLE Container Client interface, which allows PL/B to be a container for OLE Insertable objects. OLE Container objects can be manipulated using the Automation interface.

The OLE container object provides support for an OLE insertable object, such as an Excel chart. The OLE container object is a placeholder on a form for an OLE insertable object. The object to be displayed can be either created at run time or created with the designer and loaded at runtime. The OLE insertable object can also be deleted or changed at runtime.

An OLE container object can only contain one OLE insertable object at a time.

Data Declaration Format:

```
<label>  CONTAINER   [CLASS={slit}]
```

> Where: The CLASS keyword is an identification string which uniquely identifies an interface for an OLE insertable object. The size of the {slit} literal is truncated to 39 characters.

PLB Verb Usage:

```
CREATE    {object}={t}:{b}:{l}:{r}[,{properties}]
DESTROY   {object}
GETPROP   {object}[,{properties}]
SETPROP   {object}[,{properties}]
EVENTREG  {object},{event}[,{routine}[,{keyword=dest}]]
EVENTINFO {pos}[,{keyword=dest}]
LISTINS   {collection},{object}...
LISTDEL   {collection},{object}...
{object}.{method} GIVING {result} USING [{param}][,..]
```

2.7 OLE Container Object Designer Support

The PL/B form designer provides the ability to create a CONTAINER object, place an OLE insertable object within it, modify the properties of the object and browse the objects type library for methods and properties.

To create a CONTAINER object, the OleContainer tool must be selected. A sizing rectangle must then be drawn to indicate the position and size of the control. At this point the standard ‘Insert Object’ dialog is brought up, and can be either cancelled, or used to place an initial OLE insertable object in the control.

If the dialog is cancelled the CONTAINER object is created empty. When empty, the object context menu will contain an Insert Object entry, allowing the standard ‘Insert Object’ dialog to be invoked.

If the CONTAINER object is not created empty, it will be placed in a visual editing state to allow initial editing. When this editing is complete, the object context menu will contain the following entries:

- Browse Object

  This menu command will bring up an Automation browser window to allow the user to examine any Automation interface that the OLE insertable object might have.

- Insert Object

  This menu command can be used to invoke the standard ‘Insert Object’ dialog, allowing the user to replace the OLE insertable object.

- Delete Embedded Object

  This menu command will delete the OLE insertable object and make the CONTAINER object empty.

- Object Verb Set

  Below a separator line will be placed all the supported verbs for this OLE insertable object.

Once a CONTAINER object has been placed on the form, properties can be changed, code can be added, and it can be treated as another PL/B object.

2.8 OLE Container Object Creation

To create a CONTAINER object in PL/B, the programmer must first code a CONTAINER data statement, as follows:

```
OleData         CONTAINER
```

The next step is to actually create the CONTAINER object using the CREATE statement, as follows:

```
                CREATE          OleData=10:100:10:100
```

This will create an empty container with a top position of 10, a bottom position of 100, a left position of 10, and a right position of 100.

CONTAINER objects can also be created using any of the CONTAINER object's properties.

When you are finished using the CONTAINER object, it can be destroyed by using the DESTROY statement.

2.9 OLE Container Object Properties

The properties of an OLE container object can be accessed through the GETPROP and SETPROP statements. These statements are as follows:

```
GETPROP   {object}[,{properties}]
SETPROP   {object}[,{properties}]
```

Where:

{object}	- CONTAINER label reference.
{properties}	- The properties for an OLE container object are available as two groups. The first group is defined by the normal PL/B property set. This first group of properties is under direct control of the runtime. The second group of properties is provided only via the Automation interface for the OLE container object as defined for the OLE server application. The user must use vendor supplied documentation to define the specific properties available.

Group One (Direct/Static Properties)

See listed properties below.

Group Two (IDispatch Properties)

The properties available through the IDispatch interface specified use the following syntax:

```
*{name}={value}
@{svar}={value}
```

Where:

{name} - IDispatch property name. Property names are not case sensitive.

{svar} - Specifies a DIM variable which contains the IDispatch property name.

{value} - Values can be a svar, nvar, VARIANT object, or AUTOMATION object.

- ❖ AUTOACT=dnumnvar

  This property is the property indicating when an OLE container object should be activated. It may be one of:

  $AUTOACTDBLCLICK

  The object is automatically activated when the CONTAINER object is double-clicked or the ENTER key is pressed when the container object has the focus.

  $AUTOACTGETFOCUS

  The object is automatically activated when the CONTAINER object gets keyboard focus.

  $AUTOACTMANUAL

  The object is not automatically activated. It can be activated by using the $DoVerb method.

- ❖ AUTOVERB=dnumnvar

  This property enables or disables the automatic context menu for an OLE insertable object. It contains any verbs that the OLE insertable object provides.

- ❖ BACKSTYLE=dnumnvar

  This property defines the type of background to be used for supported objects. The BACKSTYLE can be OPAQUE, which prevents underlying objects from showing through this object. The BACKSTYLE can also be TRANSPARENT which allows underlying objects to show through the supported objects.

- ❖ BDRCOLOR=dnumnvar|color object

  This property allows the color of the BORDER to be specified by the user program. If the BDRCOLOR parameter is a <color object>, then the <color object> must be created before it is used. The <dnumnvar> value can also be used to specify the color as a RGB value or a Windows system color index. The <dnumnvar> value can be thought of as 4 byte values. When the high order byte contains a value of 0x00, then the next 3 bytes (24 bits) get interpreted as a RGB value. When the high order byte has a value of 0x80, then the lower order byte is used as an index value into the Windows System colors. The Windows system colors can be defined by the user under Windows 95 and Windows NT.

- BGCOLOR=dnumnvar|color object

  This property allows the user program to specify the color to be used for the background color. The parameter description is the same as for the BDRCOLOR property.

- BORDER[=dnumnvar]

  This property defines when an object is to have a border. If the <dnumnvar> parameter is not provided, this indicates that the border is to be used.

- CLASS=svarslit
- CLASSID=svarslit

  This property is the associated text name for the CLSID for the OLE insertable object. If set, the current OLE insertable object in a OLE container object will be replaced by a new OLE insertable object created using the given text name. If a question mark character is given as the name, the standard OLE Insert Object Dialog will be invoked.

- DISPLAYTYPE=dnumnvar

  This property is used to control how the OLE insertable object is drawn within an OLE container object. It may be one of:

  $DISPTYPECONTENT

  The content of an OLE insertable object is shown.

  $DISPTYPEICON

  The OLE insertable object is shown as an icon.

- HEIGHT=dnumnvar

  This property is a GETPROP/SETPROP only property. The height of the object can be retrieved using the GETPROP statement. The height is determined by subtracting the top object rectangle coordinate from the bottom object rectangle coordinate. The SETPROP statement can be used to change the current height of an object.

- HELPID=dnumnvar

  This property is used in conjunction with a Windows Help file provided by the user. If a program is executing with the F1HELP set to ON using a SETMODE statement, then this HELPID value is used as the help index context value for finding a given subject in the help file when an object has the focus and the F1 function key is entered.

- HOSTNAME=svarslit

  This property is used by the OLE insertable object's server application when the server application is open to provide editing of the OLE insertable object.

- LEFT=dnumnvar

  This property is a GETPROP/SETPROP only property. The LEFT property returns the left coordinate of a created objected.

- OBJECTID=dnumnvar

  This property specifies a identification number for an object. The OBJECTID value is available through the use of the EVENTINFO or EVENTREG statements to identify the object which caused an event to occur.

- OLEDROP=dnumnvar

  This property enables or disables the capability of dropping an OLE insertable object on an empty OLE container object.

- OLETYPE=dnumnvar

  This property returns the type of OLE insertable object in the OLE container object. This property is only valid for a GETPROP statement. Also, see the property OLETYPEA as a related property.The OLETYPE property will return one of:

$OLETYPEEMPTY	The OLE container object is empty.
$OLETYPEEMBED	The OLE container object contains an embedded OLE insertable object.
$OLETYPELINK	The OLE container object contains a linked OLE insertable object.

- OLETYPEA=dnumnvar

  This property is used to control the type of OLE insertable object allowed in an OLE container object. This property affects any method used to place an OLE insertable object into an OLE container object. See the property OLETYPE as a related property. It must be one of:

$OLETYPEBOTH	- Can contain either an embedded or linked OLE insertable object.
$OLETYPEEMBED	- Must contain a embedded OLE insertable object.
$OLETYPELINK	- Must contain a linked OLE insertable object.

❖ SIZEMODE=dnumnvar

This property is used to control how the OLE insertable object is displayed within an OLE container object. It may be one of:

$SIZEMODEAUTO

The OLE container object window is re-sized to the size of the OLE insertable object.

$SIZEMODECLIP

The OLE insertable object is clipped to the size of the OLE container object window.

$SIZEMODESCALE

The OLE insertable object is evenly scaled for a best fit in the OLE container object window.

$SIZEMODESTRETCH

The OLE insertable object is scaled to fully fit in the OLE container object window.

❖ SOURCEDOC=svarslit

This property is the filename name used as data for the OLE insertable object. If set, the current OLE insertable object in a OLE container object will be replaced by a OLE insertable object created from the data in the file.

❖ STYLE=dnumnvar|3DON|3DOFF|3DFLAT|3DOUT

The STYLE property specifies when an object should have a 3D look or a non-3D look. The various forms of STYLE defines that an object can have presentations as follows:

3DOFF	- Object does not have a 3D look.
3DON	- Object has a 3D sunken look.
3DFLAT	- Object has a 3D flat look.
3DOUT	- Object has a 3D outward look.

- TABID=dnumnvar

  This property specifies the location of this object in the tabbing order sequence. If the TABID property is not used, the object is assigned an internal tabbing order value which is determined by the order that objects are created and an initial TABID value specified for the window of the object. If the TABID property id is specified, then the value of the TABID determines the location of the object in the tabbing sequence. The tabbing sequence processes from lower TABID valued objects to higher TABID valued objects as the TAB key is entered.

.

- TOOLTIP=svarslit

  This property allows a user specified string to be provided in a tooltip window for an object.

- TOP=dnumnvar

  This property is a SETPROP/GETPROP only property. The property is the current top coordinate for an object.

- UPDATEOPT=dnumnvar

  This property is used to control when the content of an OLE insertable object is updated within an OLE container object. It may be one of:

  $UPDATEOPTAUTO

  The content of the OLE insertable object is updated any time the data is changed.

  $UPDATEOPTMANUAL

  The content of the OLE insertable object is only updated when the $Update method is called.

- VISIBLE=dnumnvar

  This property specifies if an object is to be made visible or invisible. If the VISIBLE property is set to $ON and the current state of the object is invisible, then this is the same as if an ACTIVATE statement without any activation routine was executed for the object. However, if the VISIBLE property is set $ON and the object is already visible, then the object remains visible and no change occurs. If the VISIBLE property is set to $OFF, this is the same as if a DEACTIVATE statement was executed for the object. The GETPROP retrieves the current VISIBLE state for the object.

❖ WIDTH=dnumnvar

  This property is a GETPROP/SETPROP only property. The width of the object can be retrieved using the GETPROP statement. The width is determined by subtracting the left object rectangle coordinate from the right object rectangle coordinate. The SETPROP statement can be used to change the current width of an object.

❖ ZORDER=dnumnvar

  This property specifies which object should be displayed on top when more than one object exists in the same space and in the same plane.

There are two properties in a WINDOW object that affect CONTAINER objects. These are:

❖ NEGBORDERS=dnumnvar

  This property is available for a WINDOW object to identify whether the runtime should negotiate border usage for in-place activation of an OLE container object.

❖ NEGMENUS=dnumnvar

  This property is available for a WINDOW object to identify whether the runtime should negotiate menu usage for in-place activation of an OLE container object.

2.10 OLE Container Object Methods

The methods of a CONTAINER object can be accessed through a new method statement type. This new statement type is the same as for AUTOMATION objects and is as follows:

```
{object}.{method} GIVING {result} USING [{param}][,..]
```

Where:

{object}	- CONTAINER label reference.
{method}	- When an object method is being used, the {method} name identifier is determined by referencing the vendor documentation. The {method} names are not case sensitive. The {method} can be specified either as a keyword form or as a {svar} variable with a leading '@' character. The acceptable syntax forms are as follows: {object}.methodname {object}.@{svar}
{result}	- The {result} is the expected return value for the {method} being executed for the specified {object}. The variable types for {result} can be {svar}, {nvar}, VARIANT object, or AUTOMATION object.

{param}	- The object method parameters identify the expected values required for the {method} of a specified {object}. The keyword name of a given {param} can be determined from the Automation vendor documentation. The {param} keyword names may or may not be used. If the {param} name is not specified, then the parameter must be placed into the {param} list at a valid position as documented for the method. The {param} can have one of the following syntax forms:
	
	{value}
	*{name}={value}
	@{svar}={value}
	
	Where:

{name}	- IDispatch property name. Parameter names are not case sensitive.
{svar}	- Specifies a DIM variable which contains the IDispatch parameter name.
{value}	- Values can be a svar, nvar, VARIANT object, or AUTOMATION object.

The CONTAINER object also has three internal methods, as indicated by the leading dollar sign. These methods are:

- ❖ $Delete

  The $Delete method deletes an OLE insertable object and frees up any memory used by the object. This method enables the programmer to explicitly delete an OLE insertable object. OLE insertable objects are automatically deleted when the CONTAINER object is destroyed or when the OLE insertable object is replaced with another OLE insertable object.

  This method takes no parameters and will return TRUE if successful. If a failure occurs it will return FALSE.

- ❖ $DoVerb

  The $DoVerb method is used to instruct an OLE insertable object to perform a specific operation, such as going into visual editing mode. This can be used to activate the OLE insertable object when the AUTOACT property is set to $AUTOACTMANUAL.

  This method takes one parameter named Verb. The value is a specific numeric verb code or one of:

  OLEPrimary (0)

  > This is the default action for the object.

  OLEShow (-1)

  > Activates the object for editing. If the application that created the object supports in-place activation, the object is activated within the OLE container control.

  OLEOpen (-2)

  > Opens the object in a separate application window. If the application that created the object supports in-place activation, the object is activated in its own window.

  OLEHide (-3)

  > For embedded objects, hides the application that created the object.

  OLEUIActivate (-4)

If the object supports in-place activation, activates the object for in-place activation and shows any user interface tools. If the object doesn't support in-place activation, the object doesn't activate, and an error occurs.

OLEInPlaceActivate (-5)

If the user moves the focus to the OLE container control, this creates a window for the object and prepares the object to be edited. An error occurs if the object doesn't support activation on a single mouse click.

OLEDiscardUndoState (-6)

Used when the object is activated for editing to discard all record of changes that the object's application can undo.

The method will return TRUE for success and FALSE for failure.

- ❖ $Update

The $Update method retrieves the current data from the application that supports the OLE insertable object and displays that data as a graphic in the OLE container object.

This method takes no parameters and will return TRUE if successful. If a failure occurs it will return FALSE.

2.11 OLE Container Object Events

The CONTAINER object supports a number of standard PL/B events through the EVENTREGISTER statement. These events are $CLICK, $DBLCLICK, $DRAGDROP, $DRAGOVER, $GOTFOCUS, $KEYPRESS, $LOSTFOCUS, $MOVE, $MOUSEDOWN, $MOUSEUP, $MOUSEMOVE, $RESIZE, and $OBJMOVE.

The CONTAINER object also supports one new event named $UPDATED. It has an event number of 22 and the RESULT field contains one of the following values:

- $UpdateOLEChanged (1)

  The object's data has changed.

- $UpdateOLEClosed (2)

  The file containing the linked object's data has been closed by the application that created the object.

- $UpdateOLERenamed (3)

  The file containing the linked object's data has been renamed by the application that created the object.

- $UpdateOLESaved (4)

  The object's data has been saved by the application that created the object.

2.12 OLE Container Object and the Registry

OLE insertable objects are listed in the registry under HKEY_CLASSES_ROOT by ProgID. The ProgID entry contains a sub-key with the value of the CLSID.

This CLSID value is used to look up CLSID under HKEY_CLASSES_ROOT/CLSID. The following standard keys can be found under the CLSID entry:

❖ InprocServer32	- Name of .dll for the automation object.
❖ LocalServer32	- Name of the .exe for the automation object.
❖ ProgID	- Program Identifier for this CLSID.
❖ Typelib	- CLSID of the type library information.
❖ Insertable	- Object can be used in an OLE container.
❖ Implemented Categories	– A series of CLSIDs indicating what this object provides. This can replace the Insertable key. The {40FC6ED3-2438-11cf-A3DB-080036F12502} GUID means insertable.
❖ DefaultIcon	- Default icon to be used if the data is to be displayed as an icon.
❖ Verb	- Verbs supported by this object.
❖ MiscStatus	- Special OLE status flags

The Typelib CLSID value is used to look up the type library information under HKEY_CLASSES_ROOT/TYPELIB. The following standard keys can be found under this entry:

❖ <langid>/win32	- Name and location of the type library.
❖ Flags	- Type library flags.
❖ Helpdir	- Location of the help files.

Day 3. ActiveX Controls

3.1 Introduction

ActiveX (or OLE) controls are self-contained executable components designed to be used in a window or Web page. ActiveX controls contain both visual elements and code. A PL/B program can use them in the same manner as any of the standard window controls.

ActiveX controls can be used in many types of applications, such as Microsoft Office, Microsoft Internet Explorer, and Microsoft Visual Basic.

There is a wide variety of available ActiveX controls, from a simple calendar control, to the Microsoft Internet Explorer which is just a complex ActiveX control.

An ActiveX control is really just a specific type of COM object. By the current definition, a control is some COM object that supports the IUnknown interface and is self-registering. However, the PL/B language requires a control to support at least the IOleObject interface and the IOleControl interface.

ActiveX controls use both the interfaces for Automation and for OLE insertable objects. In addition, the runtime supports the ambient property IDispatch, IOleControlSite, IPropertyNotifySink, and event IDispatch interfaces.

Since ActiveX control support is built on top of Automation, any ActiveX control object can be treated as an Automation object.

A collection of controls and information can be found at www.activex.com.

3.2 ActiveX Control Object

The CONTROL object is implemented as an ActiveX Control Client interface, which allows PLB/B to use ActiveX controls. Control objects can be manipulated using the Automation interface.

Data Declaration Format:

```
<label>  CONTROL   [CLASS={slit}]
```

Where: The CLASS keyword is an identification string which uniquely identifies an interface for an ActiveX control. If the CLASS identifier is not provided for the CONTROL variable declaration, then it must be supplied during the CREATE operation. If the CLASS identifier is supplied for the CONTROL variable declaration, then it can not be specified for the CREATE operation. The size of the {slit} literal is truncated to 39 characters.

PLB Verb Usage:

```
CREATE    {object}={t}:{b}:{l}:{r}[,{properties}]
DESTROY   {object}
GETPROP   {object}[,{properties}]
SETPROP   {object}[,{properties}]
EVENTREG  {object},{event}[,{routine}[,{keyword=dest}]]
EVENTINFO {pos}[,{keyword=dest}]
LISTINS   {collection},{object}...
LISTDEL   {collection},{object}...
{object}.{method} GIVING {result} USING [{param}][,..]
```

3.3 ActiveX Control Designer Support

The PL/B form designer provides the ability to use ActiveX controls in the same manner as the standard Windows controls supported by PL/B language.

To use an ActiveX control, it must first be added to the designer toolbox. This can be performed by selecting the Add Control menu item under the Tools menu item, or by performing a right click on the toolbox to invoke the context menu for the toolbox. The toolbox's context menu contains both an Add Control and a Delete Control menu item.

Once the Add Control menu item has been selected, a dialog will be displayed allowing the user to select the control to be added. This dialog will display a list of available ActiveX controls, and allow the selection of one control. The dialog also provides the option of using the object browser to examine an ActiveX control.

Once the control has been added to the toolbox, it can be used as if it was a standard designer control.

One special property provided for ActiveX controls by the designer is the (Properties) property. If this property is double-clicked, the designer will invoke the ActiveX controls internal property page. This allows the user to modify properties directly within the ActiveX control.

When an ActiveX control is placed on a form, runtime license information is automatically obtained and stored with the form. This license information can be examined by saving the form to source code.

The object browser can be invoked for any ActiveX control on the form by selecting the control and then selected the Browse menu item from the Tools menu.

3.4 Designer Automation Browser

The PL/B form designer provides a way of browsing the Automation interface of both OLE container objects and ActiveX controls. The browse dialog is invoked by selecting the object, then selecting the Browse menu item from the Tools menu.

This brings up a dialog containing a tree-view object, a text box, a help button, and an OK button. The browse dialog will be terminated when the OK button is pressed.

The tree-view object contains a hierarchical tree of the objects coclasses, enumerations, properties, methods, and events. At each level of the tree, the text box can contain specific information, and the help button will invoke help on the current item. Any information shown in the text box can be copied to the clipboard.

Enumerations are shown by name, under the >Enums< branch. Each enumeration set contains the name of the enumeration values in the set. Each enumeration value will cause the text box to display a CONST statement for the enumeration, and a help line. A sample is shown below:

```
CSC_NAVIGATEBACK   CONST   "2"

Navigate Back
```

Properties are shown by name, under the Properties branch. When a property is selected, the text box will contain the property data type expressed as VT_xxx data type. It will also indicate if the property can be used with a GetProp statement, a SetProp statement, or both. Finally a help line will be added if availiable. A sample is shown below:

```
Property: *Offline={VT_BOOL}

GetProp & SetProp

Controls if the frame is offline (read from cache)
```

Methods are shown by name, under the Methods branch. When a method is selected, the text box will contain a definition of the method and a help line. Each parameter is displayed as parameter name equals parameter data type. Optional parameters are enclosed in [] indicators. A sample is shown below:

```
{object}.Navigate [GIVING {VT_VOID}] USING {*URL=VT_BSTR}
                [, {*Flags=VT_VARIANT}] [, {*TargetFrameName=VT_VARIANT}]
                [, {*PostData=VT_VARIANT}] [, {*Headers=VT_VARIANT}]

Navigates to a URL or file.
```

Events are shown by name, under the Events branch. When an event is selected, the text box will contain a sample EVENTREG line, the event value, and parameter information. A sample is shown below:

```
EVENTREG {object}, $NewWindow2 [,{routine} [,ARG1={VT_DISPATCH}]
                                          [,ARG2={VT_BOOL}] ]
```

```
$NewWindow2 CONST "251"
ARG1=ppDisp, ARG2=Cancel
```

A new, hidden, non-navigated WebBrowser window is needed.

Throughout the browse tree, data types will be mostly expressed as VT_xxx values. However they may also be expressed as enum <name> as shown below:

```
{object}.QueryStatusWB GIVING {enum OLECMDF}
```

To find out what values that the enum OLECMDF has, the user can find the >Enums< branch, and the OLECMDF name under that branch.

The browser is not a substitute for documentation, but should be used as a quick reference tool.

3.5 ActiveX Control Object Creation

Most ActiveX control objects will be created on a PL/B form through the designer. But ActiveX control objects can also be created through the CREATE statement.

To create an ActiveX control object in PL/B, the programmer must first code a CONTROL data statement, as follows:

```
MyControl     CONTROL
```

This statement can optionally contain a class identifier, as follows:

```
MyControl     CONTROL     Class=”MSCAL.Calendar”
```

This marks the ActiveX control object as always being a certain type, and will be used in the future for compiler syntax checking.

The next step is to actually create the Automation object using the CREATE statement, as follows:

```
              CREATE      MyControl=10:100:10:100, Class=” MSCAL.Calendar”
```

This will create a calendar ActiveX control with a top position of 10, a bottom position of 100, a left position of 10, and a right position of 100.

If the class identifier was already specified on the data declaration, it is not required on the CREATE statement, and will cause an error if found. Otherwise, the class identifier is required.

If the ActiveX control is licensed, the LIC property must be provided on the CREATE statement. If it is not provided, the ActiveX control will be created on the computer containing the development version of the ActiveX control, but will fail on any computer with the standard version of the control.

The value for the LIC property can be obtained by creating the ActiveX control on the computer with the development version of the ActiveX control and getting the value of the LIC property using the GETPROP statement.

CONTROL objects can also be created using any of the CONTROL object’s properties.

When you are finished using the CONTROL object, it can be destroyed by using the DESTROY statement.

3.6 ActiveX Control Object Properties

The properties of an ActiveX control object can be accessed through the GETPROP and SETPROP statements. These statements are as follows:

```
GETPROP   {object}[,{properties}]
SETPROP   {object}[,{properties}]
```

Where:

{object} - CONTROL label reference.

{properties} - The properties for an ActiveX control object are available as two groups. The first group is defined by the normal PL/B property set. This first group of properties is under direct control of the runtime. The second group of properties is provided only via the Automation interface for the ActiveX control object as defined for the vendor. The user must use vendor supplied documentation to define the specific properties available. The PLBDSIGN browser capability can be used to get quick reference information about available ActiveX controls on a workstation.

Group One (Direct/Static Properties)

See listed properties below.

Group Two (IDispatch Properties)

The properties available through the IDispatch interface specified use the following syntax:

```
*{name}={value}
@{svar}={value}
```

Where:

{name} - IDispatch property name. Property names are not case sensitive.

{svar} - Specifies a DIM variable which contains the IDispatch property name.

{value} - Values can be a svar, nvar, VARIANT object, or AUTOMATION object.

- CLASS=svarslit
- CLASSID=svarslit

  This property is the associated text name for the CLSID for the ActiveX control object.

- HEIGHT=dnumnvar

  This property is a GETPROP/SETPROP only property. The height of the object can be retrieved using the GETPROP statement. The height is determined by subtracting the top object rectangle coordinate from the bottom object rectangle coordinate. The SETPROP statement can be used to change the current height of an object.

- HELPID=dnumnvar

  This property is used in conjunction with a Windows Help file provided by the user. If a program is executing with the F1HELP set to ON using a SETMODE statement, then this HELPID value is used as the help index context value for finding a given subject in the help file when an object has the focus and the F1 function key is entered.

HWND=nvar

  This property is a GETPROP only property. The HWND property value is a Windows window handle for the supported object.

- LEFT=dnumnvar

  This property is a GETPROP/SETPROP only property. The LEFT property returns the left coordinate of a created objected.

- LIC=svarslit

  This property allows a PLB program to retrieve or specify a valid LICENSE identifier for the CONTROL object.

- OBJECTID=dnumnvar

  This property specifies an identification number for an object. The OBJECTID value is available through the use of the EVENTINFO or EVENTREG statements to identify the object which caused an event to occur.

- TABID=dnumnvar

  This property specifies the location of this object in the tabbing order sequence. If the TABID property is not used, the object is assigned an internal tabbing order value which is determined by the order that objects are created and an initial TABID value specified for the window of the object. If the TABID property id is specified, then the value of the TABID determines the location of the object in the tabbing sequence. The tabbing sequence processes from lower TABID valued objects to higher TABID valued objects as the TAB key is entered.

.

- TOOLTIP=svarslit

  This property allows a user specified string to be provided in a tooltip window for an object.

- TOP=dnumnvar

  This property is a SETPROP/GETPROP only property. The property is the current top coordinate for an object.

- VISIBLE=dnumnvar

  This property specifies if an object is to be made visible or invisible. If the VISIBLE property is set to $ON and the current state of the object is invisible, then this is the same as if an ACTIVATE statement without any activation routine was executed for the object. However, if the VISIBLE property is set $ON and the object is already visible, then the object remains visible and no change occurs. If the VISIBLE property is set to $OFF, this is the same as if a DEACTIVATE statement was executed for the object. The GETPROP retrieves the current VISIBLE state for the object.

- WIDTH=dnumnvar

  This property is a GETPROP/SETPROP only property. The width of the object can be retrieved using the GETPROP statement. The width is determined by subtracting the left object rectangle coordinate from the right object rectangle coordinate. The SETPROP statement can be used to change the current width of an object.

- ZORDER=dnumnvar

  This property specifies which object should be displayed on top when more than one object exists in the same space and in the same plane.

3.7 ActiveX Control Object Methods

The methods of a CONTROL object can be accessed through a new method statement type. This new statement type is the same as for AUTOMATION objects and is as follows:

```
{object}.{method} GIVING {result} USING [{param}][,..]
```

Where:

{object}	- CONTROL label reference.
{method}	- When an object method is being used, the {method} name identifier is determined by referencing the vendor documentation. The {method} names are not case sensitive. The {method} can be specified either as a keyword form or as a {svar} variable with a leading '@' character. The acceptable syntax forms are as follows: {object}.methodname {object}.@{svar}
{result}	- The {result} is the expected return value for the {method} being executed for the specified {object}. The variable types for {result} can be {svar}, {nvar}, VARIANT object, or AUTOMATION object.
{param}	- The object method parameters identify the expected values required for the {method} of a specified {object}. The keyword name of a given {param} can be determined from the Automation vendor documentation. The {param} keyword names may or may not be used. If the {param} name is not specified, then the parameter must be placed into the {param} list at a valid position as documented for the method. The {param} can have one of the following syntax forms: {value} *{name}={value} @{svar}={value} Where: {name} - IDispatch parameter name. Parameter names are not case sensitive. {svar} - Specifies a DIM variable which contains the IDispatch parameter name. {value} - Values can be a svar, nvar, VARIANT object, or AUTOMATION object.

If the method fails, and an O145 error occurs, information concerning the exception error may be available by using the GETINFO EXCEPTION statement. The syntax for this statement is as follows:

```
<label>  GETINFO  EXCEPTION,{svar}
```

Where:

{svar} - Dim variable whose physical size is 490 characters and whose data is supplied by the failing control being used. The EXCEPTION data is follows:

1-10 - OBJECTID property value of object causing the problem.

11-20 - An error code identifying the error. Error codes should be greater than 1000. The value can be zero.

21-120 - A textual name of the source of the exception error. Typically, this is an application name. This field is blank if no name is available.

121-220 - A textual, human-readable description of the error intended for the customer. If no description, then this is blank.

221-480 - The fully qualified drive, path, and file name of a Help file with more information about the error. If no Help is available, then this field is blank.

481-490 - The Help context ID of the topic within the Help file. This field should be filled in if and only if the Help File field contains a filename. Otherwise, this field is zero.

3.8 ActiveX Control Object Events

The events for an ActiveX control object can be received through the EVENTREGISTER statement and information on an event can be obtained through the EVENTINFO statement. These statements are as follows:

```
EVENTREG  {object},{event}[,{routine}[,{keyword=dest}]]
EVENTINFO {pos}[,{keyword=dest}]
```

Where:

{object}	- CONTROL label reference.
{event}	- The {event} can be a {dnumnvar} value which identifies the event being processed. The user should reference the vendor documentation for the an ActiveX control object being used. The PLBDSIGN browse capability can be used to retrieve quick reference information about an ActiveX control object available on a workstation.
{routine}	- The {routine} specifies the label reference of the PL/B logic to be executed when a registered event occurs. Please note that event dispatching can only occur during EVENTCHECK and EVENTWAIT statement operations.
{keyword=dest}	- The 'dest' is a PL/B label reference to receive the 'keyword' parameter value. The 'dest' can be a nvar, svar, or VARIANT. The 'keyword' name can be any one of the following keywords found to the left of the equal sign character '=':

```
CHAR={svar}          - Always blank.
MODIFIER={nvar}      - Always zero.
OBJECTID={nvar}      - Object id value.
RESULT={nvar}        - Always zero.
TYPE={nvar}          - Event DISPID value.
ARG1={svar|nvar|Variant}
        .
        .
ARG10={svar|nvar|Variant} - Event argument values are defined for
                     events in the vendor documentation.
```

The only additions for ActiveX control support are the new ARG1 to ARG10 keywords. Each event can be dispatched with a number of arguments. These arguments can be retrieved by position using the EVENTINFO statement with a position of zero, or can be automatically placed in variables by using the ARG1 to ARG10 keywords in an EVENTREGISTER statement.

When a form contains code to handle an event, the code should use the EVENTINFO statement to obtain any event arguments.

The Automation browser in the designer can be used to examine the events for an ActiveX control, and it’s parameters.

3.9 ActiveX Controls and the Registry

ActiveX control objects are listed in the registry under HKEY_CLASSES_ROOT by ProgID. The ProgID entry contains a sub-key with the value of the CLSID.

This CLSID value is used to look up CLSID under HKEY_CLASSES_ROOT/CLSID. The following standard keys can be found under the CLSID entry:

❖ InprocServer32	- Name of .dll for the automation object.
❖ LocalServer32	- Name of the .exe for the automation object.
❖ ProgID	- Program Identifier for this CLSID.
❖ Typelib	- CLSID of the type library information.
❖ Insertable	- Object can be used in an OLE container.
❖ Control	- Object is an ActiveX control.
❖ Implemented Categories	– A series of CLSIDs indicating what this object provides. This can replace the Control key. The {40FC6ED4-2438-11cf-A3DB-080036F12502} GUID means control.
❖ ToolBoxBitmap32	- Default bitmap used in a designer toolbox.

The Typelib CLSID value is used to look up the type library information under HKEY_CLASSES_ROOT/TYPELIB. The following standard keys can be found under this entry:

❖ <langid>/win32	- Name and location of the type library.
❖ Flags	- Type library flags.
❖ Helpdir	- Location of the help files.

ActiveX controls must support self-registration by implementing the **DllRegisterServer** and **DllUnregisterServer** functions. The command regsvr32.exe can be used to invoke either of these two functions.

Day 4. Office97 and Database Development

4.1 Introduction

One of the many problems facing developers today is the integration of the applications they produce with external databases and office automation products. The maturity of both these technologies have made them widely used within most corporations.

The data produced by custom applications is now being used in company spreadsheets, sent in mail messages, and integrated into corporate databases.

To help with integration of office automation applications and external databases, we can use both the Automation and OLE container support technologies.

This lesson will deal specifically with the Microsoft Office 97 products set, and the ADO (Active Data Objects) database access.

Although we will only talk about Microsoft Office, these techniques and technologies apply to any office suite.

4.2 The Object Model

In Microsoft Office, each application's functionality is exposed through an Automation object interface. This top-level interface is typically a single object known as the **Application** object. The **Application** object represents the application itself, and all other objects for that application are below the **Application** object. The second level consists of the objects that the user would create or manipulate using the application. The third, fourth, and fifth levels include a variety of additional objects used to access functionality that the second level objects contain. You traverse the levels to find the object you want to use.

Together this is known as the object model for the application and represents the application's functionality in terms of objects. By interacting with an application's object model, you can manipulate the application to add custom functionality, automate processes, or integrate applications across networks.

A group of similar objects can be combined in the hierarchy as a collection. As an example, Microsoft Word can have a collection of document objects, since more than one document can be open within a session of Microsoft Word.

4.3 Microsoft Word Object Model

The Microsoft Word 97 object model is provided by MSWORD8.OLB, which is included when you install Microsoft Word 97. Online Help for this object model is available in VBAWRD8.HLP. The common location of both these files would be \Program Files\Microsoft Office\Office, but they can also be found in the \Office directory on the Office97 CD-ROM.

If you have a mixed environment of Word 97 and Word 95, you can still use the single object, WordBasic, through which Word 95 WordBasic macro commands can be accessed. You can obtain the type library (WB70EN32.TLB) from the Microsoft Office Developer Forum at: http://www.microsoft.com/officedev/. Help for WordBasic is available in WRDBASIC.HLP, which is included with Word 95.

The following is a sample program to create or open a word document and replace all the text:

```
WordObj     Automation
DocsObj     Automation
DocObj      Automation
RangeObj    Automation
AutoTrue    Integer         4,"0xffffffff"

.
. Start with the Word Application object
.
      Create             WordObj, Class="Word.Application"
      SetProp            WordObj,*Visible=AutoTrue
.
. Now get the documents object and create a new document
.
      GetProp            WordObj,*Documents=DocsObj
      DocsObj.Add        Giving DocObj
.
.To open instead enable the following code
.
.     DocsObj.Open       Giving DocObj Using "C:\WrdTest1.Doc"
.
. Now get the actual word document and close the documents object
.
      DocObj.Activate
      GetProp            DocObj,*Content=RangeObj
      SetProp            RangeObj,*Text="Hi there",*Bold=AutoTrue
      DocObj.SaveAs      Using "C:\WrdTest1.Doc"
      WordObj.Quit
```

The following is a sample program using WordBasic to create a word document:

Word	Automation	
True	Integer	4,"1"
False	Integer	4,"0"
AutoTrue	Integer	4,"0xffffffff"
AutoFalse	Integer	4,"0"

```
      Create                  Word,Class="Word.Basic"
      Word.AppMaximize
      Word.FileNewDefault
      Word.Formatfont         Using *Points=22, *Bold=True:
                              *Italic=True
      Word.Insert             Using "Using Word can be easy"
      Word.InsertPara
      Word.Formatfont         Using *Points=10, *Bold=AutoFalse:
                              *Italic=AutoFalse
      Word.Insert             Using "Using Word can be easy"
      Word.FileSaveAs         Using "C:\TestDoc.doc"
      Word.FileClose
      Word.FileExit
```

4.4 Microsoft Excel Object Model

The Microsoft Excel 97 object model is provided by EXCEL8.OLB, which is included when you install Microsoft Excel 97. Online Help for this object model is available in VBAXL8.HLP. The common location of both these files would be \Program Files\Microsoft Office\Office, but they can also be found in the \Office directory on the Office97 CD-ROM.

The following is a sample program to create and show some charts:

```
.
. This program uses automation to build an Excel chart
.
. Please change the file name D:\AutTest\Test1 to
. a existing scratch excel file
.
App         Automation  Class="Excel.Application"
Books       Automation
Book        Automation
Sheets      Automation
Sheet       Automation
Charts      Automation
Chart       Automation
Range       Automation
.
OptVar      Variant
VT_ERROR    EQU         10
DISP_E_PARAMNOTFOUND Integer  4,"0x80020004"
.
AutoTrue    Integer           4,"0xffffffff"
FileName1   Dim             250
.
. Lets make up a standard optional variant
.
      Create      OptVar,VarType=VT_ERROR,VarValue=DISP_E_PARAMNOTFOUND
.
. First create excel and make it visible
.
      Create            App
      SetProp           App,*Visible=AutoTrue
.
. Next lets get the workbooks collection
.
      GetProp           App,*Workbooks=Books
.
. Now open a book, this is really the spreadsheet file
.
.     Books.Open        Giving Book Using "D:\AutTest\Test1"
      Books.Add
      Books.Item        Giving Book Using 1
.
. Now find the first sheet from the sheets collection
.
```

```
      GetProp           Book,*Sheets=Sheets
      Sheets.Item       Giving Sheet Using 1

.
. Lets clear out the sheet
.
      Sheet.Range("A1", "W40").Clear
.
. Now stuff some data into the sheet
.
      SetProp           Sheet.Range( "A3", "A3"),*Value="March"
      SetProp           Sheet.Range( "B3", "B3"),*Value="12"

      SetProp           Sheet.Range( "A4", "A4"),*Value="April"
      SetProp           Sheet.Range( "B4", "B4"),*Value="8"

      SetProp           Sheet.Range( "A5", "A5"),*Value="May"
      SetProp           Sheet.Range( "B5", "B5"),*Value="2"

      SetProp           Sheet.Range( "A6", "A6"),*Value="June"
      SetProp           Sheet.Range( "B6", "B6"),*Value="11"

      SetProp           Sheet.Range( "A7", "A7"),*Value="July"
      SetProp           Sheet.Range( "B7", "B7"),*Value="16"
.
. Get a chart object and add a chart, then get the actual chart object
.

      Sheet.ChartObjects Giving Charts

      Charts.Add  Using *Left=100, *Top=10, *Width=350, *Height=250

      GetProp           Charts.Item(1),*Chart=Chart

.
. The chart wizard requires a range object to create the chart. We get
. this by using the special $IDispatch property. This will return the
. object reached after processing the object portion of the GetProp
. statement.
.
      GetProp           Sheet.Range( "A3", "B7"),*$IDispatch=Range

.
. Now some chart wizard fun. Note that when just using positional
. arguments, optional arguments to be skipped must be replaced by a
. VARIANT of type VT_ERROR with a value of DISP_E_PARAMNOTFOUND
.
.     Chart.ChartWizard Using Range, 11, OptVar, 1, 0, 1, 1:
.                       "Use by Month", "Month", "Useage in Thousands"
.
.
. We could have also coded this as:
.
      Chart.ChartWizard Using *Source=Range, *Gallery=11, *PlotBy=1:
                        *CategoryLabels=0, *SeriesLabels=1:
                        *HasLegend=1, *Title="Use by Month":
                        *CategoryTitle="Month":
```

```
                      *ValueTitle="Useage in Thousands"

.
. Now let them see the chart before we start the next one.
.
     Pause       "5"

.
. Now cleanup by deleting the chart and cleaning up the data and
. releasing the objects
.
     Charts.Delete
     Range.Clear
     Destroy           Range
     Destroy           Chart
     Destroy           Charts
.
. Now add the next set of data
.
     SetProp           Sheet.Range( "B3", "B3"),*Value="Chocolate"
     SetProp           Sheet.Range( "B4", "B4"),*Value="12"

     SetProp           Sheet.Range( "C3", "C3"),*Value="Vanilla"
     SetProp           Sheet.Range( "C4", "C4"),*Value="8"

     SetProp           Sheet.Range( "D3", "D3"),*Value="Orange"
     SetProp           Sheet.Range( "D4", "D4"),*Value="6"

.
. Get a chart object and add a chart, then get the actual chart object
.

     Sheet.ChartObjects Giving Charts

     Charts.Add  Using *Top=40, *Left=250, *Width=300, *Height=200

     GetProp           Charts.Item(1),*Chart=Chart

.
. Now get the data range object for Chart Wizard
.
     GetProp           Sheet.Range( "B3", "D4"),*$IDispatch=Range
.
. Make the chart
.
     Chart.ChartWizard Using *Source=Range, *Gallery=11, *PlotBy=2:
                       *CategoryLabels=0, *SeriesLabels=1:
                       *HasLegend=1, *Title="Use by Flavor":
                       *CategoryTitle="Flavor":
                       *ValueTitle="Useage in Barrells"

     Pause       "2"
.
. Now show the chart in print preview
.
     Chart.PrintOut    Using *From=1, *To=1, *Copies=1, *Preview=1:
                       *PrintToFile=0, *Collate=0
```

```
      Pause       "2"

      Destroy           Range
      Destroy           Chart
      Destroy           Charts

      Destroy           Sheet
      Destroy           Sheets

.
. Lets avoid the Save Changes? dialog
.
      SetProp           Book,*Saved=AutoTrue
.     App.GetSaveAsFileName Giving FileName1
.     Book.SaveAs     Using *Filename=FileName1
      Destroy           Book
      Destroy           Books
.
. Finally destroy excel
.
      App.Quit
      Destroy           App
      Stop
```

4.5 OLE Messaging Object Model

The OLE Messaging object model is provided by MDISP32.TLB, which is included when you install the client portion of Microsoft Exchange Server. It is also included in the Office97 CD-ROM under \OFFICE\WMS\WIN95. The default installation of MDISP32.TLB is the \WINDOWS\SYSTEM subdirectory.

Online Help for this object model is available in OLEMSG.HLP, which is included in the Microsoft Exchange Forms Designer and the Microsoft Solutions Development Kit, version 2.0.

The following is a sample program to send a message:

```
Session          Automation
Message          Automation
Recip            Automation
mapiTo           Integer          4,"1"


      Create                  Session, Class="MAPI.SESSION"
      Session.Logon
      Session.Outbox.Messages.Add   Giving Message

      Setprop                 Message,*Subject="Test message from PL/B"

      Message.Recipients.Add  Giving Recip

      Setprop                 Recip,*Name="Ed B",*Type=mapiTo

      Message.Update

      Trap                    SkipIt If Object
      Message.Send            Using *ShowDialog=True

SkipIt
      Trapclr                 Object
      Session.Logoff
```

4.6 Microsoft Outlook Object Model

The Microsoft Outlook 97 object model is provided by MSOUTL8.OLB, which is included when you install Microsoft Outlook 97. The common location of this file would be \Program Files\Microsoft Office\Office, but it can also be found in the \Office directory on the Office97 CD-ROM.

Online Help for this object model is available in VBAOUTL.HLP found in the \VALUPACK\MOREHELP directory on the Office97 CD-ROM.

The following is a sample program to send a message:

```
OutlApp     Automation
MailItem    Automation
RecMsg      Automation
.
olMailItem  Integer     4,"0"
recipient   Init        "Ed B"
isGoodName  Integer     4
Ans         Dim         1


. Connect to Outlook

      Create                  OutlApp,Class="Outlook.Application"

. Create a new mail message and add the recipient

      OutlApp.CreateItem      Giving MailItem Using olMailItem
      MailItem.Recipients.Add Giving RecMsg Using recipient
      RecMsg.Resolve          Giving isGoodName

      If              (IsGoodName = 0)
        Keyin           *ES,"The recipient's name was not found",Ans;
        Destroy         RecMsg
        Destroy         MailItem
        Destroy         OutlApp
        Stop
      Endif

      SetProp           MailItem,*Subject="Text mail using Automation":
                        *Body="Created by PL/B code"

      MailItem.Send

      Destroy           RecMsg
      Destroy           MailItem
      Destroy           OutlApp
```

4.7 Microsoft Access Object Model

The Microsoft Access 97 object model is provided by MSACC8.OLB, which is included when you install Microsoft Access 97. Online Help for this object model is available in ACVBA80.HLP. The common location of both these files would be \Program Files\Microsoft Office\Office, but they can also be found in the \Office directory on the Office97 CD-ROM.

The following is a sample program to open the sample Northwind database and preview a report :

```
AccessApp         Automation
DataBaseName      Init              "C:\Northwind.mdb"
acPreview         Form              "2"

      Create            AccessApp, Class="Access.Application"
      SetProp           AccessApp,*Visible=True

      AccessApp.OpenCurrentDatabase Using DataBaseName

      AccessApp.DoCmd.OpenReport    Using "Products by Category":
                                    acPreview

      AccessApp.Quit
```

4.8 Microsoft PowerPoint Object Model

The Microsoft PowerPoint 97 object model is provided by MSPPT8.OLB, which is included when you install Microsoft PowerPoint 97. The common location of this file would be \Program Files\Microsoft Office\Office, but it can also be found in the \Office directory on the Office97 CD-ROM.

Online Help for this object model is available in VBAPPT8.HLP found in the \Office directory on the Office97 CD-ROM.

```
PowerApp    Automation
True        Integer          4,"1"
.
. Create the link to PowerPoint and make the application visible
.
      Create      PowerApp,Class="PowerPoint.Application"
      SetProp     PowerApp,*Visible=True
.
. Show a presentation, wait 10 seconds, then close it
.
      PowerApp.Presentations.Open   Using "C:\sample.ppt"
      Pause   "10"
      PowerApp.ActivePresentation.Close

      PowerApp.Quit
      Destroy     PowerApp
```

4.9 Database Basics

At one time only large corporations would have the resources to obtain and support a large relational database. But times have changed and for a small investment, even a one-person shop can run a large scale relational database. With this change it has become important that applications be developed with the ability to both access and store data into a central database.

Microsoft has pushed the strategy of separate databases by providing limited native file access in their development products, and at the same time producing cost effective solutions using both the SQL Server and Access database products.

However, the traditional PL/B program has always used the native file capabilities of the language to store and retrieve data. This section details the options available to integrate with standard databases.

The relational model has become the standard for database design. A relational database stores and presents data as a collection of tables. Each table is then broken to columns and rows, similar to the columns and rows of a spreadsheet. Columns are also known as fields, and rows as records.

The relational model also contains a structure defining the relationships between tables to link the data in the database.

Each record contains information about one single entry in a table. Each field in a record contains a single piece of information about the record. In a customer table, information on one customer could be a customer record, and the fields of the record might be customer number, customer name, customer address, and customer contact name.

A field can be designated as a key field and index for faster retrieval. A unique key can be specified as a primary key. The primary key can be used in other tables to allow quick and exact access to a record in the table. When a key from another table is contained in a table it is known as a foreign key.

Tables can have a ‘one to many’ relationship. Each order record in an order table may be associated with only one customer record, but each customer record in a customer table may be associated to many order records.

With this terminology out of the way, the next step is a brief introduction to entity-relationship modeling, and database normalization.

Entity-relationship modeling identifies data objects or entities and the relationships between these entities. The steps are laid out below:

- Entities or objects are identified.
- Primary keys for each entity are identified.
- Data attributes or elements associated with each entity are identified. These data types should use the smallest storage space required to hold the data.
- Relationships between entities are identified, which allows the identification of primary and foreign keys.

During this process normalization also should occur. This is a set of three rules that are used to make sure that each table describes one type of entity. The rules for normalization are:

❖ First Normal Form:	You can't have multiple value columns or repeating groups.
❖ Second Normal Form:	Every non-key field must depend on the entire primary key field and not on parts of a composite primary key field.
❖ Third Normal Form:	A non-key field can't depend on another non-key field.

After this process has been completed, the resulting entity-relationship model represents a logical view of the data entities and associated relationships.

4.10 Five Second SQL

The first problem with using an external database is how to retrieve, insert, remove, and modify data. This problem is solved by the use of a standard programming syntax known as Structured Query Language (SQL).

By using SQL, a program can be written to be independent of the underlying database. This allows a company to replace one database vendor with another database vendor, and keep using the same programs developed for the previous database. This also means that a PL/B program can be written to work with a number of databases.

SQL syntax consists of an action keyword followed by any necessary parameters. Some of these keywords are:

- SELECT

  Select a series of columns from one or more tables and create a temporary result table. Records from this table can then be examined and retrieved. The overall syntax looks like:

  SELECT {column[,column][..]} FROM {table[,table][..]} WHERE {condition}

  SELECT orderid, company FROM orders, cust WHERE orders.custid=cust.custid

- DELETE

  Remove a record from a table. The overall syntax looks like:

  DELETE FROM {table} WHERE {condition}

  DELETE FROM customer WHERE custid < 200

- INSERT

  Adds a record to a table. The overall syntax looks like:

  INSERT INTO {table} ({column[,column][..]}) VALUES ({value[,value][..]})

  INSERT INTO customer (custid,company) VALUES ('ALWXX','XX')

- UPDATE

  Change one or more fields in a record or group of records. The overall syntax looks like:

  UPDATE {table} SET {column=value[,column=value][..]} WHERE {condition}

  UPDATE customer SET phone='12345678901234' WHERE custid='ALWXX'

Many of the SQL statements use a WHERE keyword and a condition to indicate the records affected. The condition is a series of Boolean expressions joined by AND, OR, and NOT keywords.

These Boolean expressions are created using columns or constants with a comparison operator. This operator can be <, >, = , <=, >=, and <>. The special keyword LIKE can also be used during a string comparison to provide a generic matching capability. The % character is used to match any number of characters, and the _ character is used to match any single character.

SQL also has a standard set of data types. Some of these types are:

❖ CHAR	- Character string of less than or equal to 255 characters
❖ LONGVARCHAR	- Character string of more than 255 characters
❖ INTEGER	- Numeric value with no decimal point
❖ DATE	- A value representing years, month, and days.
❖ TIME	- A value representing hours, minutes, and seconds.
❖ DOUBLE	- Floating point number.

4.11 ODBC

Open Database Connectivity (ODBC) provides a standard interface between databases and applications. ODBC uses database vendor drivers to access databases in much the same way that Windows uses printer drivers to access printers.

ODBC also provides a consistent API set for accessing database, and contains a unified SQL language. Both the API and SQL language have various implementation levels, allowing database drivers to written for anything from a simple flat file system, to a major relational database such as Oracle.

Programs interact with an ODBC database driver through the ODBC driver manager. The ODBC driver manager performs the following functions:

- ❖ Handles various ODBC initialization and information functions.
- ❖ Passes ODBC API calls from application to database driver, and back.
- ❖ Performs error and state checking.
- ❖ Can create a log file of the interaction between the application and the driver.

The driver manager is configured through the ODBC (32 bit) control panel. Logging can be turned on or off using this control panel, and Data Source Names (DSNs) can be configured.

A DSN is a named ODBC resource that contains the location, driver type, and other information required by an ODBC database driver to access a database. The ODBC driver uses this information to load the proper ODBC database driver, then passes the rest of the information on for driver initialization.

There are three types of DSNs. Each contains the same information, but are stored in different places. These types are:

- ❖ User DSN

  The information is stored in the Windows registry of the computer on which the DSN is created. It is only available to that specific user.

- ❖ System DSN

  The information is stored in the Windows registry of the computer on which the DSN is created. It is available to any user logged on to that computer.

- ❖ File DSN

  The information is stored in file and can be accessed from any computer.

4.12 Active Data Objects

The previous sections have discussed databases, SQL, and ODBC. This section will use all this knowledge to talk about Active Data Objects (ADO).

ADO is designed to be the application-level interface to OLE DB, Microsoft's latest and most powerful data access paradigm. OLE DB provides high-performance access to any data source such as Exchange Server or ODBC. Together ADO and OLE DB form the foundation of the Universal Data Access strategy. ADO makes it easy for developers to access the OLE DB functionality. Because ADO is built on top of OLE DB, it benefits from the rich universal data access infrastructure that OLE DB provides. ADO exists on all Windows platforms including WindowsCE.

ADO is freely available from Microsoft and can be obtained from:

www.microsoft.com/data/default.htm

The help file for ADO is named msado??.hlp, where the ?? is the version number, such as msado10.hlp.

The ADO object model consists of six objects:

❖ Connection

  The Connection object represents a connection to the data source, which can be an ODBC data source, and allows the execution of commands. The Execute method of the Connection object is used to execute any kind of command. If the command returns rows, a default Recordset object is created and returned. A more complex Recordset can be created by making a new Recordset object, associating it with the Connection, and opening the cursor. The ProgID for this object is 'ADODB.Connection'.

❖ Error

  The Error object is used to represent an error returned from a data source. This object is actually optional for the base ADO object set, since it is only needed when data sources can return multiple errors for a single method call. If a provider does not return multiple errors for a single function call, the provider just raises the error through the normal Automation mechanisms. The ProgID for this object is 'ADODB.Error'.

- Command

  The Command object represents a command (also known as a query or statement) that can be processed by the data source. Commands can return rows, but do not have to return rows. If the provider is capable, Commands can also handle parameters. The Command object is actually optional in the ADO model since some data providers cannot supply command execution.

  Commands can be simple SQL statements (or some other language the data provider recognizes) or calls to stored procedures in the database. Commands then can be executed using the Command object's Execute method. A Recordset object can also be created and associated with the Command object when opening the cursor.

  The Command object includes a collection of Parameter objects. If the provider can support commands with parameters, the Parameters collection will contain one parameter object for each parameter in the command.

  The ProgID for this object is 'ADODB.Command'.

- Parameters

  The Parameter object represents a parameter of a Command. Parameter objects can be created and added to the Parameters collection to avoid the expensive task of going to the system catalog to automatically populate the parameter binding information. The ProgID for this object is 'ADODB.Parameter'.

- Recordset

  Represents a set of records from a table, command object, or SQL Syntax. Can be created without any underlying Connection object. The ProgID for this object is 'ADODB.Recordset'.

- Field

  Represents a single column of data in a recordset. The Field object represents a column in a Recordset that can be used to obtain values, modify values, and learn about column metadata such as the column name, SQL type, or size. This object comes from the Fields collection contained in a Recordset object.

- Property

  A collection of values raised by the provider for ADO.

The following sample program uses ADO to create a recordset, read the field headers, then read all the data for the records.

```
Set             Automation
FieldCnt         Form            2
Num             Form            14
Pos             Integer         4
Data            Dim             50
CurIdx          Form            5

. Creating access to the 'Active Data Object' interface named 'Recordset'.

            Create              Set,Class="ADODB.Recordset"

. Using the 'Open' method, the dataset named 'SunbeltTest' is accessed to
. define the data collected as per the 'SQL' operation.  The DSN is defined
. in 'Settings\Control Panel\ODBC32' using the ODBC Data Source Administrator
. under the tab 'User DSN'.  In this case the SQL statement is accessing the
. data as retrieved from the 'customer'file specified to the SunbeltTest DSN.

             Set.Open         Using "Select * from customer","DSN=SunbeltTest"

. Retrieve the total field count created by the select statement.

             GetProp          Set.Fields,*Count=FieldCnt

. Obtain all the field names

             Move              "0" To Num
             Loop
              Move            Num To Pos
              GetProp         Set.Fields(Pos),*Name=Data
              Add             "1" To Num
              Break           If (Num = FieldCnt)
            Repeat

. Now move to the first record and read in all the data

            Set.MoveFirst

             Loop

              GetProp         Set,*EOF=Num
              Break           If (Num<>0)

              Move              "0" To Num
```

```
 Loop
  Move              Num To Pos
  GetProp            Set.Fields(Pos),*Value=Data
  Add                "1" To Num
  Break             If (Num = FieldCnt)
 Repeat

 Set.MoveNext
Repeat
```

. Finish by closing and destroying the dataset

```
Set.Close
Destroy        Set
```

Day 5. Putting It All Together

5.1 Introduction

The addition of COM, OLE, and ActiveX technologies to PL/B has been a major step for the language. It has provided a doorway through which the PL/B language can access current and future Microsoft Window's features.

With Automation, the programmer can now use the ADO database access objects, manipulate Office 97 applications, automatically create and send mail messages, or communicate with custom Visual Basic or Delphi programs.

OLE container support allows programs to show Office 97 documents on demand. It also allows programs to contain charts, spreadsheets, or movies. It provides the programmer with the ability to allow text to be entered on a form using Word97, then to write out a Word97 document.

Support for ActiveX controls allows the PL/B language to use a set of components common to the development of Web pages, Delphi, Visual Basic, PowerBuilder, and C++ applications. ActiveX controls can be purchased to perform tasks such as encryption, voice control, data acquisition, 3D modeling and much more.

5.2 What Is Missing

As a test of ActiveX control container conformance, the PL/B runtime has executed the Microsoft Verify Container control. This tests compliance with the version 1.1 Control Container Guidelines. The test produced no mandatory failures, 21 mandatory successes, and 32 optional successes.

Even with this level of conformance, some optional features of OLE/ActiveX technology have not yet been implemented. This list details the current limitations of the PL/B runtime.

- Type Safe Arrays

  If an array of VARIANT objects is used as a parameter or result, it must be passed in a safe array structure. This structure details the properties of the array and contains methods to extract the contents of the array. A future extension to PL/B will allow a VARIANT object to have additional properties and methods to allow it to be used as a safe array of values.

- OLEMISC_SIMPLEFRAME Support for ActiveX Controls

  This value is used to indicate that the control is a simple grouping of other controls and does little more than pass Windows messages to the control container managing the form. Controls of this sort require the implementation of ISimpleFrameSite on the container's site.

- Cut/Copy/Paste of OLE Insertable Objects

  The current runtime provides no support for the Cut/Copy/Paste menu items when a CONTAINER object has the focus.

- OLE Support for Restoring Links

  No support has been added for a dialog to restore broken links. If a PL/B form was created with a CONTAINER object that contained a link to a Word document, and the document is no longer in the same location, the creation of the CONTAINER object would fail.

- OCX 96 Enhancements

  When ActiveX controls started being used in many applications, and on Web pages, it became apparent that some improvements could be made to the OLE Control specifications. These changes are known as OCX96 and contain the following enhancements:

  - Inactive controls.
  - Windowless controls.
  - Drawing optimizations such as flicker-free activation and drawing.
  - Quick activation.
  - Better control sizing using designers.

  Most controls do not require a container to support any of these features. These features mostly provide containers with improved ways of handing controls.

- Print and Print Preview Support for OLE Insertable Objects

  CONTAINER objects that contain an OLE insertable object are not currently allowed in a PRTPAGE statement.

- Stream Support for ActiveX Controls

  Currently a PL/B form always asks an ActiveX control to store itself in a storage object using IPersistFile.

- Cut/Copy/Paste of ActiveX Controls in the Designer

  If a cut or copy operation is performed on an ActiveX control or on a group of objects containing ActiveX controls, the ActiveX controls will not be moved to the clipboard area.

5.3 Further Reading

The following is a list of books for further information.

General ActiveX Information

Understanding ActiveX and OLE	by David Chappel	ISBN 1-57231-216-5

Detailed ActiveX/OLE Information

Inside Distributed COM	Microsoft Press	ISBN 1-57231-849-x
Automation Programmer's Reference	Microsoft Press	ISBN 1-57231-584-9
OLE 2 Programmers Reference	Microsoft Press	ISBN 1-55615-628-6
Inside OLE	Microsoft Press	ISBN 1-55615-843-2
ActiveX Controls Inside Out	Microsoft Press	ISBN 1-57231-350-1

Office Development

Microsoft Access 97 Developer's Handbook	Microsoft Press	ISBN 1-57231-358-7
Microsoft Office 97 Developer's Handbook	Microsoft Press	ISBN 1-57231-440-0

SQL Information

SQL for Dummies	by Allen Taylor	ISBN 1-56884-336-4