Interface Guide

DYALOG

The tool of thought for expert programming

Version 13.0

Contents

CHAPTER 1

Introduction

Overview

This manual describes various interfaces between Dyalog APL and Windows.

Chapter 1 introduces the concepts of the Dyalog APL Graphical User Interface (GUI) and describes, in outline, how the system works. Chapter 2 contains a tutorial which takes you step-by-step through the implementation of a simple GUI application. Chapters 3 explains how to draw graphics using primitive graphical objects such as Poly, Bitmap and Metafile objects. Chapter 4 describes how to use toolbars, tab controls and status bars. Chapter 6 covers the important Grid object that provides a spreadsheet interface for displaying and editing tables of data and Chapters 7 and 8 describe the Multiple Document Interface (MDI) and docking. Further GUI material is provided in the WTUTOR, WTUTOR95 and WDESIGN workspaces.

Chapter 9 describes the TCP/IP interface which is implemented in the same object-oriented style. Chapter 10 explores how the TCP/IP interface is used to connect Dyalog APL to the Internet.

Chapters 11-13 describe the various ways in which Dyalog APL may communicate with other Windows applications using Component Object Model (COM) interfaces. These interfaces allow APL to act as an OLE Automation server and client, and allow you to write ActiveX controls in Dyalog APL.

Chapter 14 describes the DDE interface which is implemented using (traditional) APL shared variables. However, please note that DDE has all but been replaced by COM, and is no longer promoted as a major technology by Microsoft.

The final Chapter describes the interface to Microsoft Open Database Connectivity Drivers (ODBC) which allows you to access ODBC compliant databases from Dyalog APL.

Concepts

The Dyalog APL GUI is based upon four important concepts; **objects**, **properties**, **events** and **methods**.

Objects

Objects are instances of classes that contain information and provide functionality. Most Dyalog APL objects are GUI objects that may be displayed on the screen and with which you can interact. An example of an object is a push-button (an instance of class Button) which you may press to cause the program to take a particular action. Objects are defined in hierarchies.

Objects are also **namespaces** and may contain functions, variables, and indeed other namespaces. This allows you to store the code and data that is required by a given object *within* that object. Functions and variables stored in an object are hidden and protected from conflicts with functions and variables in the outside workspace and with those in other objects.

Properties

Each object has an associated set of properties which describe how it looks and behaves. For example, a Button has a property called Caption which defines the character string to be displayed in it. It also has a property called Type which may be Push (the button appears to move in and out when it is *pressed*), Radio (the button has two states and may be toggled on and off); and so forth.

Events

During interaction with the user, an object is capable of generating events. There are essentially two types of event, **raw** events and **object** events. **Raw** events are typically associated with a particular hardware operation. Pressing a mouse button, pressing a key on the keyboard, or moving the mouse pointer are examples of raw events. An **object** event is generated by some action that is specific to the object in question, but which may typically be achieved by a variety of hardware operations.

An example is the Select event. For a Button object, this event is generated when the user *presses* the Button. In MS-Windows, this can be done in several ways. Firstly, the user may click the left mouse button over the object. Secondly, under certain circumstances, the Select event can be generated when the user presses the Enter key. Finally, the event will occur if the user presses a "short-cut" (mnemonic) key that is associated with the Button.

Methods

Methods are effectively functions that an object provides; they are things that you may invoke to make the object do something for you. In Dyalog APL, the distinction between methods and events is tenuous, because events also make objects perform actions and you may generate events under program control. For example, a Scroll event is generated by a scrollbar when the user moves the thumb. Conversely, you can make a scrollbar scroll by generating a Scroll event. Nevertheless, the concept of a method is useful to describe functions that can only be invoked by a program and are not directly accessible to the user.

Objects

The following objects are supported.

System Objects

Clipboard	Provides access to Windows clipboard
Printer	For hard-copy output
Root	System-level object

Container Objects

CoolBand	Contains a single child object in a CoolBar
CoolBar	Contains resizable bands (CoolBands)
Form	Top-Level Window
Group	A frame for grouping Buttons and other objects
MDIClient	Container for MDI windows
PropertyPage	Contains a page of controls
PropertySheet	Contains PropertyPages
Splitter	Manages other objects
Static	A frame for drawing and clipping graphics
StatusBar	Ribbon statusbar
SubForm	Acts as an MDI window or a constrained Form
TabBar	Contains TabBtns (tabs)
TabControl	Windows 95/NT Tab control
ToolBar	Ribbon toolbar
ToolControl	Windows 95/NT toolbar

Menu

MenuBar	Pulldown menubar
Menu	Popup menu
MenuItem	Selects an option or action
Separator	Separator between items

Action

Button	Selects an option or action
Locator	Graphical (positional) input device
Scroll	Scrollbar
SysTrayItem	An icon in the System Tray
TabBtn	Selects a tabbed SubForm
TabButton	Selects a tabbed SubForm or performs an action
Timer	Generates events at regular intervals
TrackBar	Slider control
ToolButton	Selects an option or action
UpDown	Spin button control

Information

Animation	Displays an AVI clip
Label	Displays static text
MsgBox	Displays a message box
ProgressBar	Indicates the progress of a lengthy operation
StatusField	Displays status information
TipField	Displays pop-up context sensitive help

Input and Selection

BrowseBox	Prompts user to select a directory etc.
Calendar	Displays a month calendar
ColorButton	Allows user to select a colour
Combo	Edit field with selectable list of choices
ComboEx	Extended version of the Combo (with images)
DateTimePicker	Allows user to enter a date or time
Edit	Input field
FileBox	Prompts user to select a file
Grid	Displays a data matrix as a spreadsheet
List	For selecting an item
ListView	Displays a collection of items
RichEdit	Text field with word-processing capabilities
Spinner	Input field with spin buttons
TreeView	Displays a hierarchical set of items

Resource

Bitmap	Defines a bitmap
Cursor	Defines a cursor
Font	Loads a font
Icon	Defines an icon
ImageList	Defines a set of bitmapped images
Metafile	Loads a Windows Metafile

Graphical Output

Circle	Draws a circle
Ellipse	Draws an ellipse
Image	Displays Bitmaps, Icons and Metafiles
Marker	Draws a series of polymarkers
Poly	Draws lines
Rect	Draws rectangles
Text	Draws graphical text

Miscellaneous

ActiveXControl	Represents an APL ActiveX control
OCXClass	Provides access to OLE (ActiveX) Controls
OLEClient	Provides access to OLE Servers
OLEServer	Represents an APL OLE Server object
SM	Specifies a window for ⎕SM (character mode interface)
TCPSocket	Represents a TCP/IP socket

Implementation Overview

The Dyalog APL GUI is implemented by the following system functions :

System Function	Description
⎕WC	Creates a new object with specified properties
⎕WS	Sets the value(s) of selected properties for an existing object
⎕WG	Gets the value(s) of selected properties from an existing object
⎕WN	Reports the names of all the children of an object
⎕DQ	Waits for and processes user actions, invoking callback functions associated with events as they occur
⎕NQ	Generates an event under program control, or invokes a method

GUI Objects are a special type of *namespace* and have a name class of 9. They may therefore be managed like any other workspace object. This means that they can be localised in function headers and erased with ⎕EX. GUI objects are saved with your workspace and reappear when it is loaded or copied.

Creating Objects

You create objects using ⎕WC. Its left argument is a character vector that specifies the name of the object to be created. Its right argument specifies the object's Type and various other properties. Its (shy) result is the full pathname of the newly created object.

The following statement creates a Form called `'f1'` with the title "A Default Form" and with default size, position, etc.

```
'f1' ⎕WC 'Form' 'A Default Form'
```
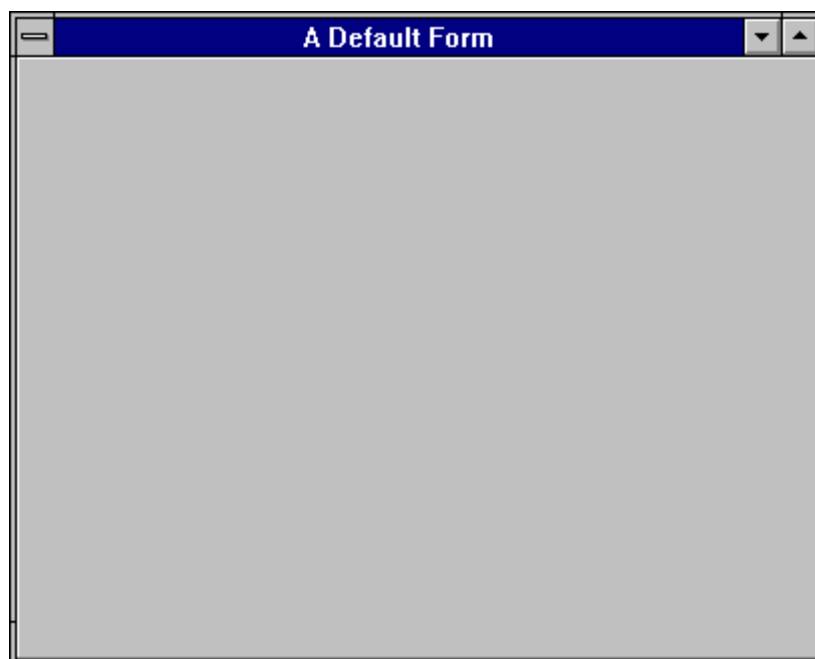
Naming Objects

Objects are created in a hierarchy. The Form we have just created is a "top-level" object to which we can attach other child objects, like buttons, scrollbars and so forth. You can create any number of top-level objects like this, up to a limit imposed by MS-Windows and your system configuration.

For reasons which will become apparent later, there is a single Root object whose name is `'.'` (dot) or `'#'`. It acts a bit like the root directory in a DOS file structure, and is the implied parent of all the top-level objects you create.

When you create a top-level object, you don't actually have to specify that it is a child of the Root; this is implied. For any other object, you specify its position in the hierarchy by including the names of its "parent", "grand-parent", and so forth in its name.

Object names are specified in the form:

```
'grandparent.parent.child'
```

where the "." character is used to separate the individual parts of the name. There is no explicit limit to the depth of the object hierarchy; although in practice it is limited by the rules governing which objects may be children of which others.

Complete object names must be unique, although you could use the same sub-name for two objects that have different parents. For example, it would be valid to have `'form1.btn1'` and `'form2.btn1'`.

Apart from the "." separator, names may include any of the characters A-Z, a-z, and 0-9. They **are** case-sensitive, so `'Form1'` is not the same name as `'form1'`.

For graphical objects, it is permissible to omit the last part of the name, although the parent name must be specified followed by a "." (dot). Further information is given later in this chapter.

Specifying Properties

The right argument of `⎕WC` is a list of properties for the object being created. Apart from trivial cases, it is always a nested vector. The first item in the list must specify the object's Type. Other properties take default values and need not always be defined. Properties are discussed more fully in the next section.

Saving Objects

Like functions, variables and operators, GUI objects are **workspace objects** and are
)SAVEd with it. GUI Objects are also *namespaces* and they have a name-class of 9.
The expression)OBJECTS or ⎕NL 9 may be used to report their names. Like other
namespaces, GUI objects may be copied from a saved workspace using)COPY or ⎕CY.

The Object Hierarchy

This example illustrates how an object hierarchy is defined. The following statements
create a Form called 'accounts' which contains a Group called 'type' and some
Buttons called 'PLAN', 'BUDGET' and 'ACTUAL'. The embedded spaces in these
statements are intended only to improve clarity. The numbers refer to the object's
position within its parent. This will be discussed in detail later.

```
'accounts'              ⎕WC 'Form' 'Accounts'
'accounts.type'         ⎕WC 'Group' 'Account Type'
'accounts.type.PLAN'    ⎕WC 'Button' 'PLAN'   (20 35)
'accounts.type.BUDGET'  ⎕WC 'Button' 'BUDGET' (45 30)
'accounts.type.ACTUAL'  ⎕WC 'Button' 'ACTUAL' (70 32)
```
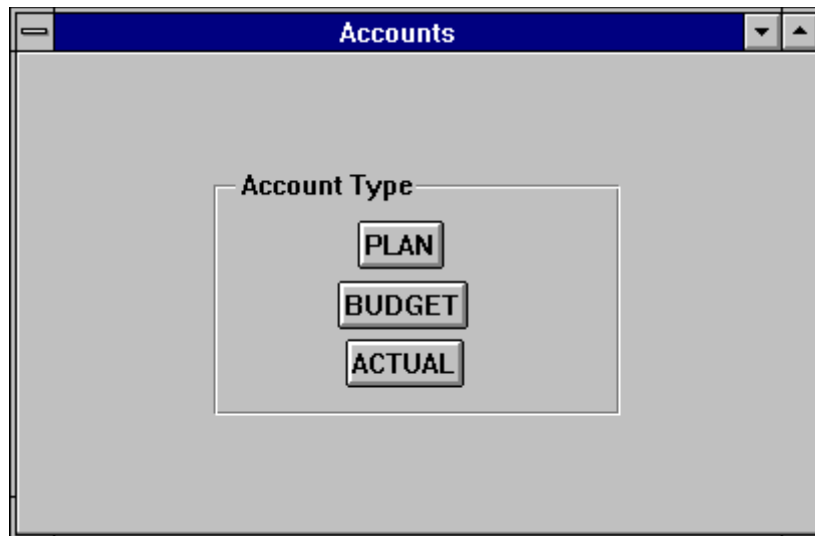

Schematically, this object structure looks as follows:

```
                            __ACTUAL
    .___accounts___type___BUDGET
                            __PLAN
```

Properties

Properties may be set using the system functions ⎕WC and ⎕WS and their values may be retrieved using ⎕WG.

If the system variable ⎕WX is set to 1, properties may be set using assignment and referenced by name as if they were variables. This is generally faster and more convenient than using ⎕WS and ⎕WG.

Certain properties, in particular the Type property, can only be set using ⎕WC. There is no obvious rule that determines whether or not a property can only be set by ⎕WC; it is a consequence of the Windows API.

However, any property that can be set by ⎕WS can be set using assignment and the values of all properties can be retrieved by direct reference or using ⎕WG.

Setting Properties with Assignment

You may set the value of a property using the assignment arrow ←. For example:

```
'F' ⎕WC 'Form'
```

The following statement sets the Caption property to the string "Hello World":

```
F.Caption←'Hello World'
```

Strand assignment may be used to set several properties in a single statement:

```
F.Size F.Posn←(40 50)(10 10)
```

However, distributed assignment is even more concise:

```
F.(Size Posn)←(40 50)(10 10)
```

Normal namespace path rules apply, so the following are all equivalent:

```
#.F.Caption←'Hello World'

)CS F
#.F
Caption←'Hello World'
```

```
:With 'F'
    Caption←'Hello World'
    Posn←40 50
    Size←10 10
    ...
:EndWith
```

Notice however, that used directly in this way, Property names are case-sensitive. The following expressions assign values to *variables* in F and have no effect on the Caption property.

```
F.caption←'Hello World'
F.CAPTION←'Hello World'
```

Retrieving property values by reference

You may obtain the value of a property as if it were a variable, by simply referring to the property name. For example:

```
F.Caption←'Hello World'

F.Caption
Hello World
```

You can retrieve the values of several properties in one statement using strand notation:

```
    F.Caption F.Posn F.Size
 Hello World  40 50  10 10
```

Although, once again, the use of parentheses is even more concise:

```
    F.(Caption Posn Size)
 Hello World  40 50  10 10
```

Although setting and referencing a Property appears to be no different to setting and referencing a variable, it is not actually the same thing at all. When you set a Property (whether by assignment or using ⎕WC or ⎕WS) to a particular value you are making a request to Windows to do so; there is no guarantee that it will be honoured. For example, having asked for a Font with face name of "Courier New", you cannot change its Fixed property to 0, because the Courier New font is always fixed pitch.

```
    'F'⎕WC'Font' 'Courier New'
1
    F.Fixed←0
    F.Fixed
1
```

Setting Properties with ⎕WC

Properties may also be set by the right argument of ⎕WC. In these cases, they may be specified in one of two ways; either by their position in the argument, or by a keyword followed by a value. The keyword is a character vector containing the **name** of the property. Its value may be any appropriate array. Property names and value keywords are not case sensitive; thus `'Form'` could be spelled `'form'`, `'FORM'`, or even `'fOrM'`

The Type property, which specifies the type of the object, applies to **all** objects and is **mandatory**. It is therefore the first to be specified in the right argument to ⎕WC, and is normally specified without the Type keyword. The value associated with the Type property is a character vector.

With the exception of Type, all other properties have default values and need only be specified if you want to override the defaults. For example, the following statements would give you a default Button in a default Group in a default Form :

```
'form'       ⎕WC 'Form'
'form.g'     ⎕WC 'Group'
'form.g.b1' ⎕WC 'Button'
```

Properties are specified in a sequence chosen to put the most commonly used ones first. In practice, this allows you to specify most properties by position, rather than by keyword/value pairs. For example, the Caption property is deemed to be the "most used" property of a Button and is specified second after Type. The following two statements are therefore equivalent:

```
'F1.B1' ⎕WC 'Button' 'OK'
'F1.B1' ⎕WC 'Button' ('Caption' 'OK')
```

The third and fourth properties are (usually) Posn, which specifies the position of a child within its parent, and Size which specifies its size. The following statements all create a Form with an empty title bar, whose top left corner is 10% down and 20% across from the top left corner of the screen, and whose height is 60% of the screen height and whose width is 40% of the screen width.

```
'form' ⎕WC 'Form' '' (10 20) (60 40)
'form' ⎕WC 'Form' '' ('Posn' 10 20) ('Size' 60 40)
'form' ⎕WC 'Form' '' ('Posn' 10 20) (60 40)
'form' ⎕WC 'Form' ('Posn' 10 20) (60 40)
```

Changing Property Values with ⎕WS

Once you have created an object using ⎕WC, you are free to alter most of its properties using ⎕WS. However in general, those properties that define the overall structure of an object's window cannot be altered using ⎕WS. Such *immutable* properties include Type and (for some objects) Style. Note that if you find that you do need to alter one of these properties dynamically, it is a simple matter to recreate the object with ⎕WC.

The syntax for ⎕WS is identical to that of ⎕WC. The following examples illustrate how the properties of a Button can be altered dynamically. Note that you can use ⎕WS in a callback function to change the properties of any object, including the one that generated the event.

Create "OK" button at (10,10) that calls FOO when pressed

```
 'form.b1' ⎕WC 'Button' 'OK' (10 10)
```

Some time later, change caption and size

```
 'form.b1' ⎕WS ('Caption' 'Yes') ('Size' 20 15)
```

Note that if the right argument to ⎕WS specifies a single property, it is not necessary to enclose it.

How the Property List is Processed

The system is designed to give you as much flexibility as possible in specifying property values. You should find that any "reasonable" specification will be accepted. However, you may find the following explanation of how the right argument of ⎕WC and ⎕WS is parsed, useful. **The casual reader may wish to skip this page**.

Items in the right argument are processed one by one. If the next array in the argument is a simple array, or a nested array whose first element is not a character vector, the array is taken to be the value of the next property, taking the properties in the order defined for that object type.

When the system encounters a **nested array** whose first element is a character vector, it is checked against the list of property names. If it is not a property name, the entire array is taken to define the value of the next property as above.

If the first element **is** a property name, the remainder of the nested array is taken to be the value of the corresponding property. For convenience, considerable latitude is allowed in how the structure of the property value is specified.

After assigning the value, the parser resets its internal pointer to the property following the one named. Thus in the third and fourth examples on the preceding page, omitting the Size keyword is acceptable, because Size is the next property after Posn.

In the reference section for each object, you will find the list of properties applicable to that object, given in the order in which they are to be specified. This information is also reported by the PropList property, which applies to all objects. The list of properties may also be obtained by executing the system command)PROPS in an object's namespace.

The Event Property

Of the many different properties supported, the Event property is rather special. Most of the other properties determine an object's appearance and general behaviour. The Event property, however, specifies how the application **reacts to the user**. Furthermore, unlike most other properties, it takes not a single value, but a set of values, each of which determines the action to be taken when a particular **event** occurs. In simple terms, an *event* is something that the user can do. For example, pressing a mouse button, pressing a key, selecting an item from a menu, are all examples of *events*.

Like any other property, the Event property may be set by assignment or using ⎕WC and ⎕WS. Using assignment, you can specify settings for the entire set of events, or you can set individual events one by one.

Each type of event has a *name* and a *number*. Although you may identify an event either by its name or by its number, the use of its name is generally preferable. The exception to this is user-defined events which may only be specified by number.

The list of events supported by a particular object is available from its EventList property, or by executing the system command)EVENTS in an object's namespace.

To specify an individual event, you assign the action to the event name which is optionally prefixed by the string 'on' . For example, the name for the event that occurs when a user presses a key is 'KeyPress'. To this you assign an *action*. Event *actions* are described in detail later in this chapter, but most commonly *action* is a character vector containing the name of a function. This is termed a *callback* function, because it will be automatically *called* for you when the corresponding event occurs. So if F1 is a Form, the statement:

```
F1.onKeyPress←'CHECK_KEY'
```

specifies that the system is to call the function CHECK_KEY whenever the user presses a key when F1 has the input focus.

Using ⎕WC and ⎕WS, the same effect can be obtained by:

```
'F1'⎕WC'Form' ('Event' 'onKeyPress' 'CHECK_KEY')
```
or
```
'F1'⎕WS 'Event' 'onKeyPress' 'CHECK_KEY'
```

When a callback function is invoked, the system supplies an *event message* as its right argument, and (optionally) an array that you specify, as its left argument. The event message is a nested vector that contains information about the event. The first element of the event message is always either a *namespace reference* to the object that generated the event or a character vector containing its name.

To instruct the system to pass the object *name* instead of a *reference*, you must use the event name on its own (omitting the `'on'` prefix) or the event number. This method is retained for compatibility with previous versions of Dyalog APL that did not support namespace references. For example, either of the following statements will associate the callback function `'CHECK_KEY'` with the KeyPress event. However, when `'CHECK_KEY'` is called, it will be called with the character string `'F1'` in the first element of the right argument (the event message) instead of a direct reference to the object `F1`.

```
F1.Event←'KeyPress' 'CHECK_KEY'
'F1'⎕WS 'Event' 'KeyPress' 'CHECK_KEY'
'F1'⎕WS 'Event' 22 'CHECK_KEY'
```

Note that by default, all events are processed automatically by APL, and may be ignored by your application unless you want to take a specific action. Thus, for example, you don't have to handle Configure events when the user resizes your Form; you can just let APL handle them for you.

Before looking further into events, it is necessary to describe how control is passed to the user, and to introduce the concept of the *event queue*.

For further details, see the description of the Event property in the *Object Reference*.

User Interaction & Events

Giving Control to the User

As we have seen, `⎕WC` and `⎕WS` are used to build up the definition of the user-interface as a hierarchy of **objects** with **properties**. Notice that the interface is defined not only in terms of its appearance and general behaviour, but also by specification of the Event property, in terms of how it reacts to user actions.

Once you have defined your interface, you are ready to give control to the user. This is simply done by calling `⎕DQ`. Alternatively, you may use the Wait method (if appropriate) which is identical to `⎕DQ` in its operation.

`⎕DQ` performs several tasks. Firstly, it displays all objects that have been created but not yet drawn. When you create objects, Dyalog APL/W automatically buffers the output so as to avoid unpleasant flashing on the screen. Output is flushed when APL requires input (at the 6-space prompt) and by `⎕DQ`. Thus if you write a function that creates a Form containing a set of controls, nothing is drawn until, later on in the function, you call `⎕DQ`. At this point the Form and its contents are displayed in a single screen update, which is visually more pleasing than if they were drawn one by one. A second task for `⎕DQ` is to cause the system to wait for user events. Objects that you create are immediately active and capable of generating events. During development and testing, you can immediately use them without an explicit *wait*. However, unless your application uses the **Session** in conjunction with GUI objects you must call `⎕DQ` to cause the application to wait for user input. In a run-time application, `⎕DQ` is **essential**.

The right argument to `⎕DQ` specifies the objects with which the user may interact. If it specifies `'.'`, the user may interact with **all** active objects owned by the current thread **and** with any new objects which are created in callback functions. If not, the right argument is a simple character vector or a vector of character vectors, containing the names of one or more Form or PropertySheet objects and the Clipboard object, or the name of a single *modal* object of type FileBox, Locator, MsgBox or Menu. All specified objects must be owned by the current thread.

In general, `⎕DQ` first updates the screen with any pending changes, then hands control to the user and waits for an event. If its right argument is `'.'` `⎕DQ` processes events for all active objects, i.e. for those objects and their children whose Active property is 1. If the right argument contains the name of one or more Form and/or Clipboard objects, `⎕DQ` processes events for all of these objects and their children, and (if the current thread is thread 0) for the Root object, but ignores any others, even though they may be currently active.

If the right argument specifies a single modal object, `⎕DQ` displays the object on the screen, handles user-interaction with it, and then hides the object when the user has finished with it. An event is generated according to the manner in which the user terminated.

Events are managed by both the Operating System and by `⎕DQ` using a **queue**. A detailed understanding of how the queue works is not absolutely necessary, and you may skip the following explanation. However, if you are planning to develop major applications using the GUI, please continue.

The Event Queue

There are in fact two separate queues, one maintained by MS-Windows and one internal to APL. The MS-Windows queue is used to capture all events that APL needs to process. These include events for your GUI objects as well as other events concerned with APL's own Session Window, Edit Windows, etc. At various points during execution, APL reads events from the MS-Windows queue and either processes them immediately or, if they are events concerned with objects you have defined with `⎕WC`, APL places them on its own internal queue. It is this queue to which `⎕DQ` looks for its next event.

When `⎕DQ` receives an event, it can either ignore it, process it internally, execute a string, call a callback function, or terminate according to the action you have defined for that event. The way you define different actions is described in detail later in this Chapter.

If you have disabled a particular event by setting its action code to ‾1, `⎕DQ` simply ignores it. For example, if you set the action code of a KeyPress event to ‾1, keystrokes in that object will be ignored. If you have told `⎕DQ` to process an event normally (the default action code of 0) `⎕DQ` performs the default processing for the event in question. For example, the default processing for a KeyPress event in an Edit object is to display the character and move the input cursor.

If you have associated a string or a callback function with a particular event in a particular object, `⎕DQ` executes the string or invokes the callback function for you. During the execution of the string or the callback function, the user may cause other events. If so, these are added to APL's internal queue but they are not acted upon immediately. When the execution of the string or the callback function terminates, control returns to `⎕DQ` which once more looks to the internal queue. If another event has been added while the callback function was running, this is read and acted upon. If not, `⎕DQ` looks to the MS-Windows queue and waits for the next event to occur.

If you have associated an **asynchronous** callback function with an event (by appending the character "&" to the name of the function), ⎕DQ starts the callback function in a new thread and is then immediately ready to process the next event; ⎕DQ does not wait for an asynchronous callback function to complete.

If ⎕DQ reads an event with an associated action code of 1 , it terminates and returns the **event message** which was generated by the event, as a result. The normal processing for the event is not actioned. During the time between ⎕DQ terminating and you calling it again, events are discarded. Events are only stored up in the queue if ⎕DQ is active (i.e. there is a ⎕DQ in the state indicator). It is therefore usually better to process events using callback functions.

Assignment and reference to the Event Property

There are a number of special considerations when using assignment and reference to the Event property.

You can set the action for a single event by prefixing the Event name by "on". For example, to set the action of a MouseUp event on a Form F to execute the callback function FOO:

```
      F.onMouseUp←'UP'
      F.onMouseUp
 #.UP
```

Notice that the value returned (#.UP) is not necessarily exactly the same as you set it (UP).

If you reference the Event property, you will obtain all the current settings, reported in order of their internal event number. Notice the use of distributed strand notation to set more than one event in the same statement.

```
      F.(onMouseUp onMouseDown)←'UP' ('DOWN' 42)
      F.Event
 onMouseDown  #.DOWN  42    onMouseUp  #.UP
```

If you set the Event property using assignment, all the event actions are redefined, i.e. previous event settings are lost. For example:

```
      F.(onMouseUp onMouseDown)←'UP' ('DOWN' 42)
      F.Event
 onMouseDown  #.DOWN  42    onMouseUp  #.UP

      F.Event←'onMouseMove' 'MOVE'
      F.Event
 onMouseMove  #.MOVE
```

The All event can also be set by assignment, and it too clears previous settings. Notice too that a subsequent reference to a specific event using the "on" prefix, will report the "All" setting, unless it is specifically reset.

```
      F.(onMouseUp onMouseDown)←'UP' ('DOWN' 42)
      F.Event
 onMouseDown  #.DOWN  42   onMouseUp  #.UP

      F.onAll←'FOO'
      F.Event
 onAll  #.FOO

      F.onMouseMove
 #.FOO

      F.Event←'onMouseMove' 'MOVE'
      F.Event
 onMouseMove  #.MOVE
```

If no events are set, the result obtained by ⎕WG and the result obtained by referencing Event directly are different:

```
      'F'⎕WC'Form'
      DISPLAY 'F'⎕WG'Event'
.→--.
|0 0|
'~--'
      DISPLAY F.Event
.⊖-----------.
| .→--------. |
| | .⊖. .⊖. | |
| | |_| |_| | |
| '∊--------' |
'∊-----------'
```
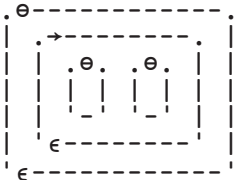
Callback Functions

By setting the action code to 1 for all the events you are interested in, you could write the control loop in your application as:

```
Loop:   Event ← ⎕DQ 'system'
        test Event[1] (object name)
        and  Event[2] (event code)
        →Label

Label: process event for object
        →Loop
```

However, such code can be error prone and difficult to maintain. Another limitation is that events that occur between successive calls on ⎕DQ are discarded.

An alternative is to use callback functions. Not only do they encourage an object-oriented modular approach to programming, but they can also be used to validate the user's actions and prevent something untoward happening. For example, a callback function can prevent the user from terminating the application at an inappropriate point. The use of callback functions will also produce applications that execute faster than those that process events by exiting ⎕DQ and looping back again as above.

You associate a callback function with a particular event or set of events in a given object. There is nothing to prevent you from using the same callback function with several objects, but it only makes sense to do so if the processing for the event(s) is common to all of them. The object that caused the event is identified by the first element of the right argument when the callback is invoked.

When an event occurs that has an action set to a character vector, the system looks for a function with that name. If none exists ⎕DQ terminates with a **VALUE ERROR**. If the function does exist, it is called. If the callback function was called **FOO** and it stopped on line [1], the State Indicator would be:

```
      )SI
FOO[1]*
⎕DQ
...
```

A callback function may be defined with any syntax, i.e. it may be dyadic, monadic, or niladic. If it is monadic or dyadic, ⎕DQ calls it with the event message as its right argument. If the function is dyadic, its left argument will contain the value of the array that was associated with the event.

A callback function is otherwise no different from any other function that you can define. Indeed there is nothing to prevent you from calling one explicitly in your code. For example, a callback function that is invoked automatically could call a second callback function directly, perhaps to simulate another event.

By default, a callback function is run synchronously. This means that ⎕DQ waits for it to return a result before attempting to process any other events. Events that are generated by Windows while the callback function is running are simply queued.

Alternatively, you may specify that a callback function is to be run **asynchronously**. In this case, ⎕DQ starts the function in a new thread, but instead of waiting for it to complete, proceeds immediately to the next event in the queue. See Asynchronous Callbacks for further information.

Modifying or Inhibiting the Default Processing

It is often desirable to inhibit the normal processing of an event, and it is occasionally useful to substitute some other action for the default. One way of inhibiting an event is to set its action code to ¯1. However this mechanism is non-selective and is not always applicable. You can use it for example to ignore **all** keystrokes, but not to ignore particular ones.

Synchronous callback functions provide an additional mechanism which allows you to selectively inhibit default processing of an event. The mechanism also allows you to modify the event in order to achieve a different effect.

For example, you can use a callback function to ignore a **particular** keystroke or set of keystrokes, or even to replace the original keystroke with a different one. Similarly, you can use a callback function to selectively ignore a LostFocus event if the data in the field is invalid. Callback functions therefore give you much finer control over event processing. The mechanism uses the result returned by the callback function and operates as follows.

When an event occurs that has a synchronous callback function attached, ⎕DQ invokes the callback function (passing it the event message as its right argument) before performing any other action and waits for the callback to complete. When the callback function terminates (exits) ⎕DQ examines its result.

If the callback function returned no result, or returned a scalar 1 or the identical event message with which it was invoked, ⎕DQ then carries out the default processing for the event in question. If the callback function returned a 0, ⎕DQ takes no further action and the event is effectively ignored. Finally, if the callback returns a **different** event message (from the one supplied as its right argument), ⎕DQ performs the default processing associated with the new event rather than with the original one.

For example, consider a callback function attached to a KeyPress event in an Edit object. When the user presses a key, for the sake of example, the unshifted "a" key, ⎕DQ invokes the callback function, passing it the corresponding event message as its right argument. This event message includes information about which key was pressed, in this case "a". The various possibilities are:

- If the callback function returns a value of 1 or the same event message with which it was invoked, ⎕DQ carries out the default processing for the original event. In this case a lower-case "a" is displayed in the field.

- If the callback function returns a value of 0, ⎕DQ takes no further action and the keystroke is ignored.

- If the callback function modifies the event message and changes the key from an "a" to a "b", ⎕DQ carries out the default processing associated with the *new* event, and displays a lower-case "b" instead.

Note that asynchronous callback functions may not be used to modify or inhibit the default processing because their results are ignored.

Generating Events using □NQ

The □NQ system function is used to generate events under program control and has several uses.

Firstly, it can be used to do something automatically for the user. For example, the following expression gives the input focus to the object Form1.ED1.

```
□NQ 'Form1.ED1' 'GotFocus'
```

Secondly, □NQ can be used to generate user-defined events which trigger special actions either by invoking callback functions or by causing □DQ to terminate. For example, if you were to define the Event property on 'Form1' as:

```
'Form1' □WS ('Event' 1001 'FOO')('Event' 1002 1)
```

The expression:

```
□NQ 'Form1' 1001 'Hello' 42
```

would cause □DQ to invoke the function FOO, passing it the entire event message ('Form1' 1001 'Hello' 42) as its right argument. Similarly, the expression:

```
□NQ 'Form1' 1002 23.59
```

would cause □DQ to terminate with the array ('Form1' 1002 23.59) as its result.

□NQ can be used to generate events in one of three ways which affect the **context** in which the event is processed.

If it is used monadically as in the examples above, or with a left argument of 0, □NQ adds the event specified in its right argument onto the bottom of the event queue. The event is then processed by □DQ when it reaches the head of the queue. You can add events to the queue **prior** to calling □DQ, or from within a callback function which is itself called by □DQ. In either case, the context in which the event is finally processed may be completely different from the context in which the event was placed on the queue. When used in this way, the result of □NQ is always an empty character vector.

If you use ⎕NQ with a left argument of 1, the event is processed there and then by ⎕NQ itself. If there is a callback function attached to the event, ⎕NQ invokes it directly. Thus like ⎕DQ, ⎕NQ can appear in the State Indicator ⎕SI or)SI. This use of ⎕NQ is used to generate an event for an object that is not currently included in a ⎕DQ, and is the usual way of generating the special (non-user) events on the Printer and other objects. It is also used when you want to cause an event to occur **immediately** without waiting for any events already in the queue to be processed first. When used in this way, the result of ⎕NQ is either an empty character vector, or the result of the callback function if one is attached.

If you use ⎕NQ with a left argument of 2, APL immediately performs the default processing (if any) for the event, bypassing any callback function. This case of ⎕NQ is often used *within* a callback function to put the object into the state that it would otherwise be in when the callback terminated. When used in this way, the result of ⎕NQ is 1. To avoid processing the event twice, the callback function should return 0.

The use of ⎕NQ with a left argument of 2, is the same as calling the event as a method, and this is discussed in the next section.

A left argument of 4 is a special case that is used by an ActiveXControl or NetType object to generate an event in its host application. See Chapter 13 for details.

Methods

Calling Methods

A method is similar to a function in that it may or may not take an argument, perform some action, and return a result.

Examples are the Print, NewPage, Setup and Abort methods, all of which cause a Printer object to take a particular action.

If the system variable `⎕WX` is 1, you may invoke an object's method using exactly the same syntax as you would use to call a function in that object.

For example, to execute the IDNToDate method of a Calendar object named `F.CAL`, you can use the expression:

```
      F.CAL.IDNToDate 36525
2000 1 1 5
```

When you call a method in this way, the method name is case-sensitive and if you spell it incorrectly, you will get a `VALUE ERROR`.

```
      F.CAL.idntodate 36525
VALUE ERROR
      F.C.idntodate 36525
     ^
```

Invoking Methods with ⎕NQ

Methods may also be called using `⎕NQ` with a left argument of 2, indeed if `⎕WX` is 0, this is the only way to call a method.

The result of the method is returned by `⎕NQ`. Note however that the result is *shy*.

For example, for a TreeView object you can obtain status information about a particular item in the object using the GetItemState method:

```
      ⎕←2 ⎕NQ 'f.tv' 'GetItemState' 6
96
```

Or you can call the IDNToDate method of a Calendar object `F.C` as follows:

```
      ⎕←2 ⎕NQ 'F.CAL' 'IDNToDate' 36525
2000 1 1 5
```

When you call a method using 2 ⎕NQ , the method name is **not** case-sensitive.

```
      ⎕←2 ⎕NQ 'F.CAL' 'idntodate' 36525
2000 1 1 5
```

Events as Methods

Methods and events are closely related and most events can be invoked as methods.

For example, you can reposition and resize a Form in one step by calling its Configure event as a method. The argument you supply is the same as the event message associated with the event, but with the first two items (Object and Event code) omitted.

```
      F.Configure 10 10 30 20
```

Or, using 2 ⎕NQ

```
      2 ⎕NQ 'F' 'Configure' 10 10 30 20
```

Notice that when you call an event as a method, you are executing the *default processing* associated with the event. The setting for the Event property is ignored and, in particular, any callback associated with the event is not executed.

GUI Objects as Namespaces

GUI objects are a special type of namespace and this has several useful implications. Firstly, instead of creating the children of an object from *outside* in the workspace, you can use `)CS` to change to an object and create them from within. The only restriction is that you can only create GUI objects that are valid as children of the current object. A second benefit is that you can put the callback functions, together with any global variables they require, into the objects to which they apply. Consider the following example.

First make a Form `F1`

```
'F1' ⎕WC 'Form' 'GUI Objects as Namespaces'
              ('Size' 25 50)
```

Then change to the Form's namespace

```
      )CS F1
#.F1
```

Now you can create a Group (or any other child object), but because you are already *inside* the Form, the name you give to the Group will be taken as being *relative* to the Form. In other words, you must specify the part of the name that applies to the Group itself, leaving out the `'F1.'` prefix that you would use if you executed the statement *outside* in the workspace.

```
      'CH' ⎕WC 'Group' 'Counter' (10 10)( 70 60)
```

You can continue to create other objects

```
      'OK'  ⎕WC 'Button' '&Ok' (20 80)(θ 15)
      'CAN' ⎕WC 'Button' '&Cancel' (60 80) (θ 15)
      'FNT' ⎕WC 'Font' 'Arial' 16 ('Weight' 700)
```

If you ask for a list of objects, you will see only those within the current namespace

```
      )OBJECTS
CAN     CH      FNT     OK
```

When you are inside an object you can also set (or get) a property directly, so you can set the FontObj property for the Form with the following statement.

```
      FontObj←'FNT'
```

You can achieve the same with `⎕WS` by omitting its left argument:

```
⎕WS 'FontObj' 'FNT'
```

You can create a child of the Group from outside it ...

```
'CH.UP' ⎕WC 'Button' '+1' (20 10)(30 20)
```

or you can change to it and create others from within...

```
      )CS CH
#.F1.CH
      'DOWN' ⎕WC 'Button' '-1' (60 10)(30 20)
      'FNT' ⎕WC 'Font' 'Arial' 32
      'CTR' ⎕WC 'Label' ('FieldType' 'Numeric' )
              ('FontObj' 'FNT')
```

Once again, if you request a list of objects you will see only those in the current namespace.

```
      )OBJECTS
CTR     DOWN    FNT     UP
```

You can create functions and variables in a GUI namespace in exactly the same way as in any other. So, for example, you could create a variable called `COUNT` and a function `CHANGE` to update it:

```
      COUNT ← 0

      ∇ INCR CHANGE MSG
[1]     COUNT←COUNT+INCR
[2]     CTR.Value←COUNT
      ∇
```

You can also make `CHANGE` a callback function for the two Buttons.

```
      UP.onSelect←'CHANGE' 1
      DOWN.onSelect←'CHANGE' ¯1
```

Notice that because you were in the `F1.CH` namespace when you made this association, the event will fire the function `CHANGE` in the `F1.CH` namespace and, furthermore, it will execute it within that namespace. Hence the names referenced by the function are the local names, i.e. the variable `COUNT` and the Label `CTR`, within that namespace.

So if you now switch back to the outer workspace

```
      )CS
#
```

and click on the buttons...

The result will appear approximately as shown below
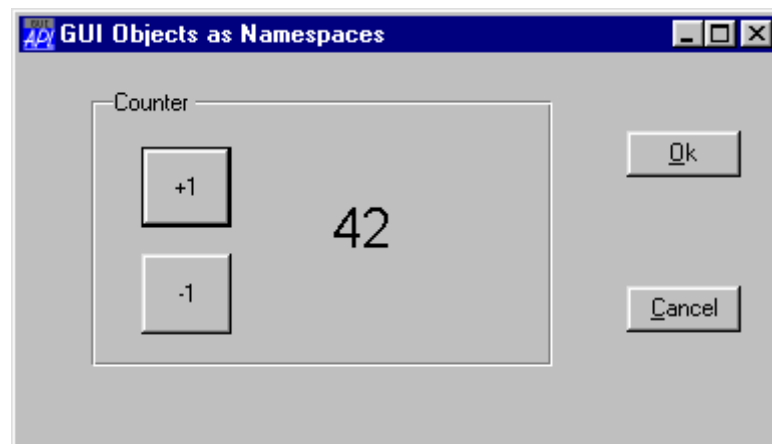
Attaching GUI Objects to Namespaces

Monadic `⎕WC` is used to *attach* a GUI component to an existing object. The existing object must be a pure namespace or an appropriate GUI object (one that can legitimately be placed at that point in the object hierarchy). The operation may be performed by changing space to the object or by running `⎕WC` *inside* the object using the *dot* syntax. For example, the following statements are equivalent.

```
      )CS F
#.F
      ⎕WC 'Form'  ⍝ Attach a Form to this namespace


      )CS
#
      F.⎕WC'Form' ⍝ Attach a Form to namespace F
```

Monadic `⎕WC` is often used in conjunction with the KeepOnClose property. This property specifies whether or not an object remains in existence when its parent Form (or in the case of a Form, the Form itself) is closed by the user or receives a Close event.

This facility is particularly useful if you wish to have functions and variables encapsulated within your Forms. You may want to save these structures in your workspace, but you do not necessarily want the Forms to be visible when the workspace is loaded.

An alternative way to achieve this is to prevent the user from closing the Form and instead make it invisible. This is achieved by intercepting the Close event on the Form and set its Visible property to 0. Then, when the Form is subsequently required, its Visible property is set back to 1. However, if the Form needs adjustment because the workspace was loaded on a PC with different screen resolution or for other reasons, it may not be easy to achieve the desired result using `⎕WS`. Monadic `⎕WC` is generally a better solution.

Namespace References and GUI Objects

The use of a GUI name in an expression is a reference to the GUI object, or *ref* for short. If you assign a *ref* or call a function with a *ref* as an argument, the *reference* to the GUI object is copied, not the GUI object itself.

So for example, if you have a Form named `F`:

```
      'F'⎕WC 'Form'
```

Assigning `F` to `F1`, does not create a second Form `F1`; it simply creates a second reference (`F1`) to the Form `F`. Subsequently, you can manipulate the Form `F` using either `F` or `F1`.

```
      F1←F
      F1
#.F
      F1.Caption←'Hello World'
      F.Caption
Hello World
```

Similarly, if you call a function with `F` as the argument, the local argument name becomes a second reference to the Form, and a new Form is not created:

Here is a simple function which approximately centres a Form in the middle of the screen:

```
      ∇ R←SHOW_CENTRE FORM;OLD;SCREEN
[1]     SCREEN←⊃'.'⎕WG'DevCaps'
[2]     OLD←FORM.Coord
[3]     FORM.Coord←'Pixel'
[4]     R←FORM.Posn←⌊0.5×SCREEN-FORM.Size
[5]     FORM.Coord←OLD
      ∇
```

The function can be called using either F or F1 (or any other Form) as an argument:

```
      SHOW_CENTRE F
287 329
      SHOW_CENTRE F1
287 329
```

A *ref* to a GUI object can conveniently be used as the argument to `:With`; for example, the `SHOW_CENTRE` function can instead be written as follows:

```
      ∇ R←SHOW_CENTRE FORM;OLD;SCREEN
[1]    SCREEN←⊃'.'⎕WG'DevCaps'
[2]    :With FORM
[3]        OLD←Coord
[4]        Coord←'Pixel'
[5]        R←Posn←⌊0.5×SCREEN-Size
[6]        Coord←OLD
[7]    :EndWith
      ∇
```

If instead, you actually want to duplicate (clone) a GUI object, you may do so by calling `⎕WC` with a *ref* as the right argument and the new name as the left argument.

For example:

```
      'F' ⎕WC 'Form' 'Cloning Example'
      'F.B' ⎕WC 'Group' 'Background' (10 10)(80 30)
      'F.B.R' ⎕WC 'Button' 'Red' (20 10)('Style' 'Radio')
      'F.B.B' ⎕WC 'Button' 'Blue' (50 10)('Style' 'Radio')
      'F.B.G' ⎕WC 'Button' 'Green' (80 10)('Style' 'Radio')
```

Then, instead of creating a second Group for selecting Foreground colour line by line as before, you can clone the "Background" Group as follows:

```
      'F.F' ⎕WC F.B
```

The new Group `F.F` is an exact copy of `F.B` and will have the same Posn, Size and Caption, as well as having identical children. To achieve the desired result, it is therefore only necessary to change its Posn and Caption properties; for example:

```
      F.F.Caption F.F.Posn ← 'ForeGround' (10 60)
```

The result is illustrated below.
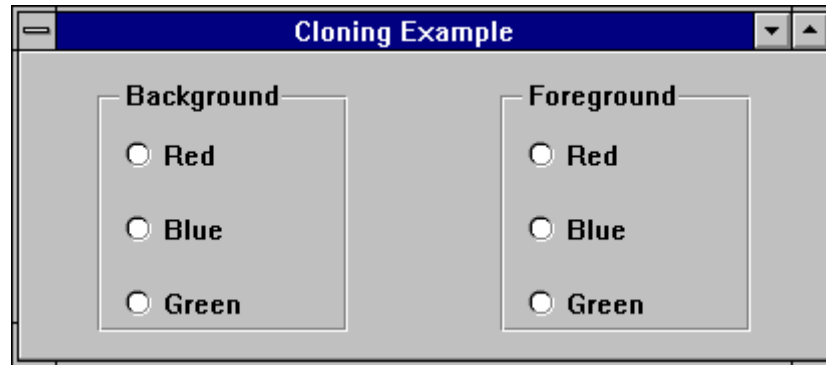
Note that when a namespace is cloned in this way, the objects (functions, variables and other namespaces) within it are not necessarily duplicated. Instead, the objects in cloned namespaces are in effect just pointers to the original objects. However, if you subsequently change the clone, or the original object to which it refers, the two are de-coupled and a second *copy* ensues. This mechanism makes it possible to create large numbers of *instances* of a single *class* namespace without consuming an excessive amount of workspace.

Modal Dialog Boxes

Up to now, it has been assumed that your user has constant access to all of the interface features and controls that you have provided. The user is in charge; your application merely responds to his requests.

Although this is generally considered desirable, there are times when a particular operation must be allowed to complete before anything else can be done. For example, an unexpected error may occur and the user must decide upon the next course of action (e.g. Continue, Restart, Quit). In these situations, a *modal* dialog box is required. A modal dialog box is one to which the user must respond before the application will continue. While the modal dialog box is in operation, interaction with all other objects is inhibited.

A modal dialog box is simply achieved by calling `⎕DQ` with just the name of the corresponding Form in its argument. This can be done from within a callback function or indeed from any point in an application. To make the local `⎕DQ` terminate, you may specify an action code of 1 for an event. Alternatively, if you wish to make exclusive use of callback functions to process events, you can cause the `⎕DQ` to terminate by erasing the Form from a callback function.

For example, suppose that you want the user to close the dialog box by clicking an "OK" button. You would specify the Event property for the Button as:

```
('Event' 'Select' 'EXIT')
```

... and the function `EXIT` is simply...

```
      ∇ EXIT Msg;BTN;Form
[1]    ⍝ Terminate modal ⎕DQ by erasing Form
[2]     OBJ←⊃Msg
[3]     Form←(¯1+OBJι'.')↑OBJ ⍝ Get Form name
[4]     ⎕EX Form
      ∇
```

Note that this function is fairly general, as it gets the name of the Form from the name of the object that generated the event.

The MsgBox and FileBox Objects

The MsgBox and FileBox objects are standard MS-Windows dialog boxes and are strictly modal. The following discussion refers to the way a MsgBox is used, but applies equally to a FileBox.

The MsgBox is a pop-up modal dialog box with a title bar (defined by the Caption property), an icon (defined by the Style property), some text (defined by the Text property) and up to three buttons (defined by the Btns property).

The MsgBox does not appear on the screen when you create it with ⎕WC. Instead, it pops up ONLY when you call ⎕DQ with the name of the MsgBox as its sole right argument. Furthermore, the MsgBox automatically pops down when the user clicks on any one of its buttons; you don't actually have to enable any events to achieve this. For example:

```
'ERR' ⎕WC 'MsgBox' 'Input Error' '' 'Error'
```

creates an invisible MsgBox with the title (Caption) `'Input Error'`, no text, and a Style of `'Error'`. This gives it a "Stop sign" icon. When you want to issue an error message to your user, you simply call a function (let's call it ERRMSG) which is defined as follows:

```
      ∇ ERRMSG Msg
[1]    ⍝ Displays 'ERR' message box
[2]     ERR.Text←Msg ⍝ Put Msg in box
[3]    ⎕DQ 'ERR'
      ∇
```

Note that ⎕DQ will terminate automatically when the user clicks one of the buttons in the MsgBox object.

In this case we were not interested in the particular button that the user pressed. If you are interested in this information, you can enable special events associated with these buttons. For details, see the description of the MsgBox and FileBox objects in the Object Reference.

Multi-Threading with Objects

The following rules apply when using threads and objects together.

1. All events generated by an object are reported to the thread that *owns* the object and cannot be detected by any other threads. A thread *owns* an object if it has created it or inherited it. If a thread terminates without destroying an object, the ownership of the object and its events passes to the parent thread.

2. The Root object '.' and the Session object ⎕SE are owned by thread 0. Events on these objects will be only be detected and processed by ⎕DQ running in thread 0, or by the implicit ⎕DQ that runs in the Session during development.

3. Several threads may invoke ⎕DQ concurrently. However, each thread may only use ⎕DQ on objects that it owns. If a thread attempts to invoke ⎕DQ on an object owned by another thread, it will fail with DOMAIN ERROR.

4. Any thread may execute the expression ⎕DQ '.', however:

    4.1. In thread 0, the expression ⎕DQ '.' will detect and process events on the Root object and on any Forms and other top-level objects owned by thread 0 or created by callbacks running in thread 0. The expression will terminate if there are no active and visible top level objects **and** there are no callbacks attached to events on Root.

    4.2. In any other thread, the expression ⎕DQ '.' will detect and process events on any Forms and other top-level objects owned by that thread or created by callbacks running in that thread. The expression will terminate if there are no active and visible top level objects owned by that thread.

5. A thread may use ⎕NQ to *post* an event to an object owned by another thread, or to invoke the default processing for an event, or to execute a method in such an object. This means that the following uses of ⎕NQ are allowed when the object in question is owned by another thread:

    ```
      ⎕NQ object event...
    0 ⎕NQ object event...
    2 ⎕NQ object event...
    2 ⎕NQ object method...
    3 ⎕NQ ole_object method...
    4 ⎕NQ activexcontrol event...
    ```

    The only use of ⎕NQ that is prohibited in these circumstances is

    ```
    1 ⎕NQ object event...
    ```

    which will generate a DOMAIN ERROR.

6. While a thread is waiting for user response to a *strictly modal* object such as a MsgBox, FileBox, Menu or Locator object, any other threads that are running remain suspended. APL is not able to switch execution to another thread in these circumstances.

The Co-ordinate System

Each object has a Coord property that determines the units in which its Posn and Size properties are expressed. Coord may be set to one of the following values :

`'Inherit'`	this means that the object assumes the same co-ordinate system as its parent. This is the default for all objects except the Root object.
`'Prop'`	the position and size of the object are expressed as a percentage of the dimensions of its parent.
`'Pixel'`	The position and size of the object are expressed in pixels.
`'User'`	the position and size of the object are expressed in units defined by the YRange and XRange properties of the object's parent.
`'Cell'`	the position and size of the object are expressed in cell coordinates (applies only to Grid and its graphical children).

By default, the value of Coord for the Root object is `'Prop'`. For all other objects, the default is `'Inherit'`. This means that the default co-ordinate system is a proportional one.

You can change Coord from one value to another as you require. It only affects the units in which Size and Posn are currently expressed. The physical position and size are unaffected. Note that if you set Posn and/or Size in the same ⎕WC or ⎕WS statement as you set Coord, it is the **old** value of Coord that is applied.

The co-ordinate system is also independent of the way in which objects are reconfigured when their parent is resized. This is perhaps not immediately obvious, as it might be expected that objects which are specified using Pixel co-ordinates will be unaffected when their parent is resized. This is not necessarily the case as the manner in which objects respond to their parent being resized is determined **independently** by the AutoConf and Attach properties.

The User co-ordinate system is useful not only to automate scaling for graphics, but also to perform scrolling. This is possible because XRange and YRange define not just the scale along each axis, but also the position of the origin of the co-ordinate system in the parent window.

Colour

Colours are specified using the FCol (foreground colour) and BCol (background colour) properties. Graphical objects have an additional FillCol (fill colour) property.

A single colour may be specified in one of two ways, either as a negative integer that refers to one of a set of standard Windows colours, or as a 3-element numeric vector. The latter specifies a colour directly in terms of its red, green and blue intensities which are measured on the scale of 0 (none) to 255 (full intensity). Standard Windows colours are:

Colour	Element	Colour	Element
0	Default	¯11	Active Border
¯1	Scroll Bars	¯12	Inactive Border
¯2	Desktop	¯13	Application Workspace
¯3	Active Title Bar	¯14	Highlight
¯4	Inactive Title Bar	¯15	Highlighted Text
¯5	Menu Bar	¯16	Button Face
¯6	Window Background	¯17	Button Shadow
¯7	Window Frame	¯18	Disabled Text
¯8	Menu Text	¯19	Button Text
¯9	Window Text	¯20	Inactive Title Bar Text
¯10	Active Title Bar Text	¯21	Button Highlight

A colour specification of 0 (which is the default) selects the appropriate background or foreground colour defined by your current colour scheme for the object in question. For example, if you select yellow as your MS-Windows Menu Bar colour, you will get a yellow background in Menu and MenuItem objects as the default if BCol is not specified.

To select a colour explicitly, you specify its RGB components as a 3-element vector. For example:

```
(255   0   0) = red        (  0 255   0) = green
(255 255   0) = yellow     (192 192 192) = grey
(  0   0   0) = black      (255 255 255) = white
```

Note that the colour **actually realised** depends upon the capabilities of the device in question and the current contents of the Windows colour map.

A colour specification of θ (zilde) selects a transparent colour.

Fonts

In keeping with the manner in which fonts are managed by Microsoft Windows and other GUI environments, Dyalog APL treats fonts as objects which you create (load) using ⎕WC and erase (unload) using ⎕EX or localisation.

A Font **object** is created and assigned a name using ⎕WC. This name is then referenced by other objects via their FontObj **properties**. For example to use an Arial bold italic font of height 32 pixels in two Labels:

```
'A32' ⎕WC 'Font' 'ARIAL' 32 0 1 0 700

'F.L1' ⎕WC 'Label' 'Hello' (20 10) ('FontObj' 'A32')
'F.L2' ⎕WC 'Label' 'World' (20 10) ('FontObj' 'A32')
```

If a font is referenced by more than one Form, you should create the Font as a top-level object, as in the above example. However, if the font is referenced by a single Form, you may make the Font object a child of that Form. The font will then automatically be unloaded when you erase the Form with which it is associated.

Compatibility Note:

In the first release of Dyalog APL/W (Version 6.2), fonts were referenced **directly** by the FontObj property. The above example would have been achieved by:

```
'F.L1' ⎕WC 'Label' 'Hello' (10 10)
            ('FontObj' 'ARIAL' 32 0 1 0 700)

'F.L2' ⎕WC 'Label' 'World' (20 10)
            ('FontObj' 'ARIAL' 32 0 1 0 700)
```

Although this original mechanism continues to be supported, it is recommended that you use the method based on Font **objects** which supersedes it.

Drag and Drop

Dyalog APL/W provides built-in support for drag/drop operations through the Dragable property. This applies to all objects for which drag/drop is appropriate.

The default value of Dragable is 0 which means that the object cannot be drag/dropped. To enable drag/drop, you can set it to 1 or 2. A value of 1 means that the user drags a box that represents the bounding rectangle of the object. In general, a value of 2 means that the user drags the outline of the object itself, whether or not it is rectangular. However, there are two exceptions. For a Text object, (`'Dragable' 2`) means that the user drags the text itself. For an Image object that contains an Icon, (`'Dragable' 2`) means that the user drags the icon itself, and not just its outline.

If Dragable is 1 or 2, the user may drag/drop the object using the mouse.

When the user drops an object, the default processing for the event is:

a) If the object is dropped over its parent, it is moved to the new location.

b) If the object is dropped over an object other than its parent, the dragged object remains where it is.

If you enable the DragDrop event (11) on all eligible objects, you can control what happens explicitly. If an object is dropped onto a new parent, you can move it by first deleting it and then recreating it. Note that you must give it a new name to reflect its new parentage. Note too that the DragDrop event reports co-ordinates relative to the object being dropped **on**, so it is easy to rebuild the object in the correct position and with the correct size.

An alternative to using the built-in drag/drop operation is to do it yourself with the Locator object. This is no less efficient and has the advantage that you can choose which mouse button you use. It can also be used to move a group of objects. However, the Locator only supports a rectangular or elliptical outline.

Debugging

Four features are built into the system to assist in developing and debugging GUI applications.

Firstly, if you execute `⎕WC` and/or `⎕WS` statements from the Session or by tracing a function, they have an **immediate** effect on the screen. Thus you can see immediately the visual result of an expression, and go back and edit it if it isn't quite what you want.

Secondly, if you use `⎕WC` with an existing name, the original object is destroyed and then re-created. This allows you to repeatedly edit and execute a single statement until it gives the effect you desire.

Thirdly, if you TRACE a `⎕DQ` statement, any callback functions that are invoked will be traced as they occur. This is invaluable in debugging. However, callbacks invoked by certain "raw" events, for example MouseMove, can be awkward to trace as the act of moving the mouse pointer to the Trace window interferes with the operation in the object concerned.

Finally, `⎕NQ` can be used to artificially generate events and sequences of events in a controlled manner. This can be useful for developing repeatable test data for your application.

Creating Objects using ⎕NEW

With the introduction of Classes in Version 11.0, you may manipulate Dyalog GUI objects as Instances of built-in (GUI) Classes. This approach supplements (but does not replace) the use of ⎕WC, ⎕WS and so forth.

To create a GUI object using ⎕NEW, the Class is given as the GUI Object name and the Constructor Argument as a vector of (Property Name / Property Value) pairs. For example, to create a Form:

```
F1←⎕NEW 'Form' (⊂'Caption' 'Hello World')
```

Notice however that only perfectly formed name/value pairs are accepted. The highly flexible syntax for specifying Properties by position and omitting levels of enclosure, that is supported by ⎕WC and ⎕WS, is not provided with ⎕NEW.

Naturally, you may reference and assign Properties in the same way as for objects created using ⎕WC:

```
      F1.Size
50 50
      F1.Size←20 30
```

Callbacks to regular defined functions in the Root or in another space, work in the same way too. If function FOO merely displays its argument:

```
      ∇ FOO M
[1]    ⎕←M
      ∇
```

```
      F1.onMouseUp←'#.FOO'
#.[Form]  MouseUp  78.57142639 44.62540...
```

Note that the first item in the event message is a ref to the Instance of the Form.

To create a control such as a Button, it is only necessary to run ⎕NEW inside a ref to the appropriate parent object. For example:

```
      B1←F1.⎕NEW 'Button' (('Caption' '&OK')('Size' (10 10)))
```

As illustrated in this example, it is not necessary to assign the resulting Button Instance to a name *inside* the Form (`F1` in this case). However, it is a good idea to do so that refs to Instances of controls are expunged when the parent object is expunged. In the example above, expunging `F1` will not remove the Form from the screen because `B1` still exists as a ref to the Button. So, the following is safer:

```
F1.B1←F1.⎕NEW'Button'(('Caption' '&OK')('Size' (10 10)))
```

Or perhaps better still,

```
F1.(B1←⎕NEW 'Button'(('Caption' '&OK')('Size' (10 10))))
```

CHAPTER 2

GUI Tutorial

Introduction

This tutorial illustrates how to go about developing a GUI application in Dyalog APL/W. It is necessarily an elementary example, but illustrates what is involved. The example is a simple Temperature Converter. It lets you enter a temperature value in either Fahrenheit or Centigrade and have it converted to the other scale.

Some Concepts

Objects

Objects are GUI components such as Forms, Buttons and Scrollbars. You create objects using the system function ⎕WC. Its left argument is a name for the object; its right argument specifies the object type and various properties. Objects are created in a hierarchy.

Properties

Properties specify the appearance and behaviour of an object. For example, the Caption property specifies the text that appears on a Button or the title that appears in the title bar on a Form. When you create an object with ⎕WC, its right argument specifies its properties. You can also set properties using ⎕WS. This lets you dynamically alter the appearance and behaviour of an object as required.

Events

Events are things that happen in objects as a result (usually) of some action by the user. For example, when the user clicks a MenuItem, it generates a Select event. Similarly, when the user focuses on an object, it generates a GotFocus event.

Callback Functions

Callback Functions are APL functions that you can associate with a particular event in a particular object. Interaction with the user is controlled by the system function ⎕DQ. This function performs all of the routine tasks involved in driving the GUI interface. However, its main role is to invoke your callback functions for you as and when events occur.

That's enough theory for now ... let's see how it all works in practice.

Creating a Form

The first task is to create a Form which is to act as the main window for our application. We will call the Form 'TEMP' and give it a title (Caption) of "Temperature Converter".

We will position the Form 68% down and 50% along the screen. This will avoid it interfering with the APL Session Window, and make development easier.

The Form will have a height equal to 30% of the height of the screen, and a width of 40% of the screen width.

```
TITLE←'Temperature Converter'
'TEMP' ⎕WC 'Form' TITLE (68 50)(30 40)
```
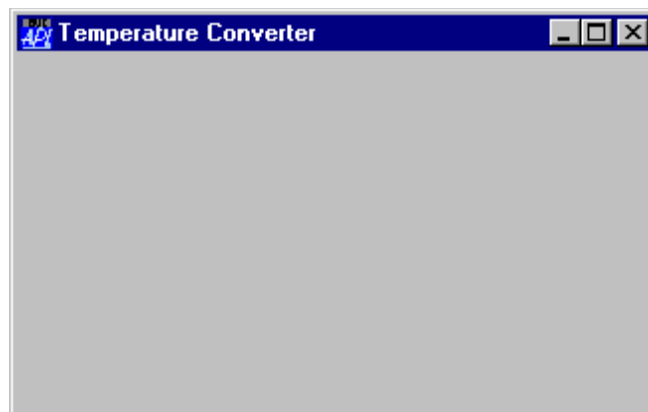
Adding a Fahrenheit Label

We are going to need two edit fields to input and display temperatures and two labels to identify them.

Let's create the "Fahrenheit" label first. It doesn't really matter what we call it because we won't need to refer to it later. Nevertheless, it has to have a name. Let's call it `LF`. We will place it at (10,10) but we don't need to specify its Size; `⎕WC` will make it just big enough to fit its Caption.

```
'TEMP.LF' ⎕WC'Label' 'Fahrenheit'(10 10)
```
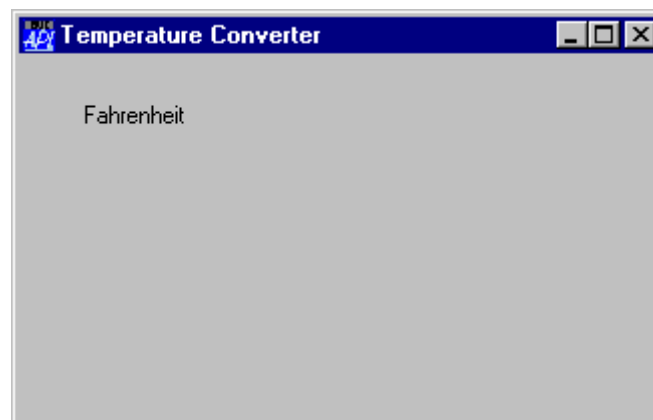
Adding a Fahrenheit Edit Field

Now let's add the edit field for Fahrenheit. We will call it F and place it alongside the label, but 40% along. Initially the field will be empty. We will make it 20% wide but let its height default. ⎕WC will make it just big enough to fit the current font height. As the field is to handle numbers, we will set its FieldType to 'Numeric'.

```
'TEMP.F' ⎕WC 'Edit' '' (10 40)(0 20)('FieldType' 'Numeric')
```
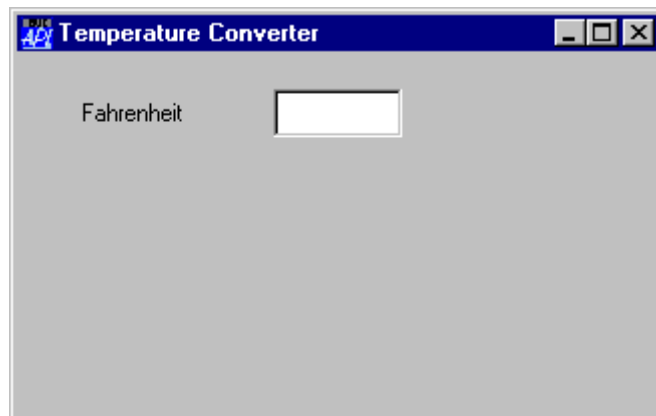
Adding a Centigrade Label & Edit Field

Now we need to add a corresponding Centigrade label and edit field. We'll call these
objects LC and C respectively, and place them 40% down the Form.

```
'TEMP.LC' ⎕WC'Label' 'Centigrade' (40 10)
'TEMP.C' ⎕WC 'Edit' '' (40 40)(0 20)('FieldType' 'Numeric')
```
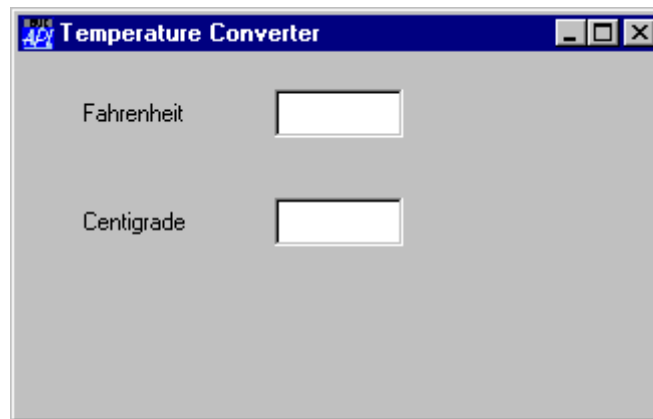
Adding Calculate Buttons

Our Temperature Converter must work both ways; from Fahrenheit to Centigrade and vice versa. There are a number of different ways of making this happen.

A simple approach is to have two buttons for the user to press; one for Fahrenheit to Centigrade, and one for Centigrade to Fahrenheit. We will call the first one `F2C` and place it alongside the Fahrenheit edit field. The caption on this button will be `'F->C'`. When the user presses the button, we want our application to calculate the centigrade temperature. For this we need a *callback function*. Let's call it `f2c`. Notice how you associate a callback function with a particular event. In this case, the Select event. This event is generated by a Button when it is pressed.

(The statement below is broken into two only so as to fit on the page)

```
    FB←'Button' 'F->C' (10 70)('Event' 'Select' 'f2c')
    'TEMP.F2C'⎕WC FB
```
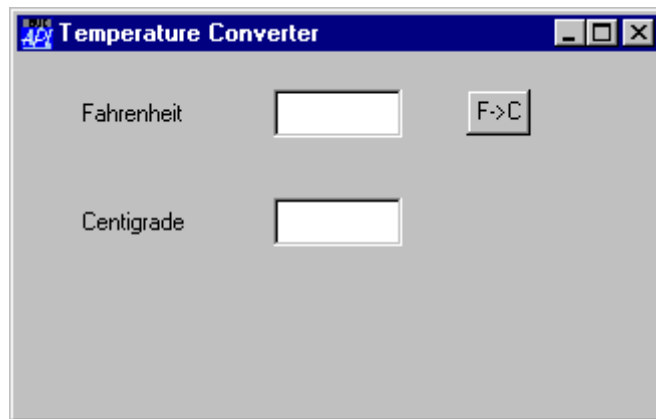

Notice that it is not necessary to specify the Size of the button; the default size fits the Caption nicely. Now let's add the Centigrade to Fahrenheit button. This will be called `C2F` and have an associated callback function `c2f`. We could have chosen to have a single callback function associated with both buttons, which would save a few lines of code. Having separate functions is perhaps clearer.

Again, the statement is split into two only to make it fit onto the page.

```
FC←'Button' 'C->F' (40 70)('Event' 'Select' 'c2f')
'TEMP.C2F'⎕WC FC
```
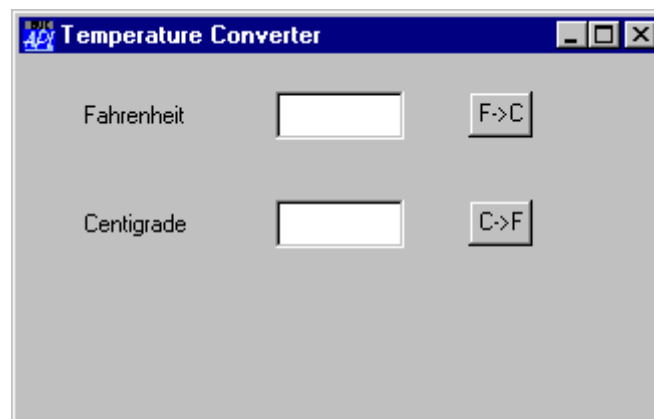

Closing the Application Window

Then we need something to allow our user to terminate our application. He will expect the application to terminate when he closes the window. We will implement this by having a callback function called `QUIT` which will simply call `⎕OFF`, i.e.

```
    ∇ QUIT
[1]   ⎕OFF
    ∇
```

We can associate this with the Close event on the Form `TEMP`. This event will be generated when the user closes the window from its System Menu

```
    TEMP.onClose←'QUIT'
```

Although here we have used assignment to set the Event property, we could just as easily have defined it when we created the Form by adding (`'Event' 'Close' 'QUIT'`) to the right argument of `⎕WC`.

Adding a Quit Button

Finally, we will add a "Quit" button, attaching the same `QUIT` function as a callback, but this time to the Select event which occurs when the user presses it.

Instead of having a default sized button, we will make it nice and big, and position it centrally.

To make the statement fit on the page, it is split into three. The Posn and Size properties are explicitly named for clarity.

```
QB←'Button' '&Quit' ('Posn' 70 30)
QB,←('Size' 0 40)('Event' 'Select' 'QUIT')
'TEMP.Q' ⎕WC QB
```
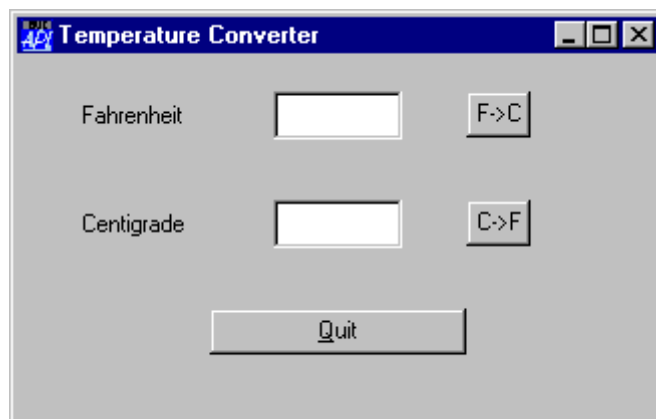

Notice how the ampersand (&) in the Caption is used to specify the *mnemonic* (short-cut) key. This can be used to press the button instead of clicking the mouse.

The Calculation Functions

So far we have built the user-interface, and we have written one callback function `QUIT` to terminate the application. We now need to write the two functions `f2c` and `c2f` which will actually perform the conversions. First let's tackle `f2c`.

A callback such as this one performs just one simple action. This does not depend upon the type of event that called it (there is only one), so the function has no need of arguments. Neither does it need to do anything fancy, such as preventing the event from proceeding. It need not therefore return a result. The header line, which includes the local variables we will need, is then...

```
[0]  f2c;F;C
```

The first thing the function must do is to obtain the contents of the Fahrenheit edit field which is called `TEMP.F`. As we have defined the FieldType as `'Numeric'`, this is easily done by querying its Value property...

```
[1]  F ← TEMP.F.Value
```

Next, we need to calculate Centigrade from Fahrenheit...

```
[2]  C ← (F-32) × 5÷9
```

... and finally, display the value in the Centigrade edit field. As we have also defined this as a numeric field, we can just set its Value property using assignment.

```
[3]  TEMP.C.Value←C
```

So our completed `f2c` callback function is...

```
     ∇ f2c;F;C
[1]  F ← TEMP.F.Value
[2]  C ← (F-32) × 5÷9
[3]  TEMP.C.Value←C
     ∇
```

which can be simplified to:

```
     ∇ f2c
[1]     TEMP.C.Value←(TEMP.F.Value-32)×5÷9
     ∇
```

The Centigrade to Fahrenheit callback function `c2f` looks very similar:

```
     ∇ c2F
[1]     TEMP.F.Value←32+TEMP.F.Value×9÷5
     ∇
```

Testing the Application

Before we test our application, it would be a good idea to `)SAVE` the workspace. If you remember, the `QUIT` callback calls `⎕OFF`, so if we don't want to lose our work...

```
      )SAVE TEMP
TEMP saved ...
```

Note that the GUI objects we have created are all saved with the workspace. You don't have to re-build them every time you `)LOAD` it again.

If this was a Run-Time application, we would have to use `⎕DQ` to run it. However, as it is not, we can just go ahead and use it from the Session. Click on the Fahrenheit edit field and enter a number (say 212). Now click on the "F->C" button. The Temperature Converter window should look like the picture below.
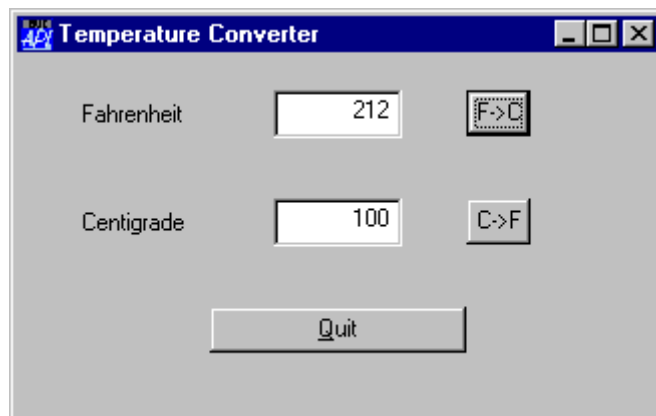
If you have mis-typed any of the functions in this example, you may get an error. If this happens, don't worry; simply correct the error as you would with any other APL application, and type →⎕LC.

If you got a `VALUE ERROR error`, you have probably mis-spelt the name of the callback function. If so, you can fix it using `⎕WS` to reset the appropriate Event property.

Don't click the "Quit" button or close the window (yet). If you do so your APL session will terminate.

If you want to follow what is happening you can use the Tracer. This requires a statement to trace, so we will use ⎕DQ just as you would in a real application. To do this, type ⎕DQ '.' in the Session window, then, instead of pressing Enter (to execute it), press Ctrl+Enter (to Trace it). Having done this, enter your data into one of the edit fields and click the "F->C" or "C->F" buttons as before. When you do so, your callback function will pop-up in a Trace Window. Step it through (if in doubt, see the section on the Tracer) and watch how it works. When the callback has finished, its Trace window disappears. Don't forget, you are running a ⎕DQ. To terminate it, press Ctrl+Break or select Interrupt from the Action menu.

Making the Enter Key Work

Ok, so the basic application works. Let's look at what we can do to improve it.

The first thing we can do is to let the user press the Enter key to make the system re-calculate, rather than having to click on a button. There are a number of alternatives, but we will do it using the Default property of Buttons.

In any Form, you can allocate a single Button to be the Default Button. In simple terms, pressing Enter anywhere in the Form has the same effect as clicking the Default Button. Let's do this for the "F->C" Button :

```
TEMP.F2C.Default←1
```

Now type a number into the Fahrenheit field and then press the Enter key. As you will see, this fires the Default Button labelled "F->C". The only problem with this is that the user cannot run the calculation the other way round using the Enter key. We need some mechanism to switch which Button is the Default one depending upon which field the user typed in.

This is easily achieved by making use of the GotFocus event. This is generated when the user puts the cursor into the edit field prior to typing. So all we need do is attach a callback to set the Default Button whenever a GotFocus event occurs in either edit field. We could use two separate callbacks or we could make use of the fact that you can make APL supply data of your choice to a callback when it is fired. This is supplied as its left argument.

The first of the next two statements attaches the callback function `'SET_DEF'` to the GotFocus event in the Fahrenheit edit field. It also specifies that when APL runs the callback, it should supply the character vector `'TEMP.F2C'` to `SET_DEF` as its left argument. `'TEMP.F2C'` is of course the name of the Button which we want to make the Default one. The second statement is identical, except that it instructs APL to supply the name of the Centigrade to Fahrenheit Button `'TEMP.C2F'`

```
      TEMP.F.onGotFocus ← 'SET_DEF' 'TEMP.F2C'
      TEMP.C.onGotFocus ← 'SET_DEF' 'TEMP.C2F'
```

Where the callback `'SET_DEF'` is defined as...

```
      ∇ BTN SET_DEF MSG
[1]    BTN □WS'Default' 1
      ∇
```

Now let's test the application again. Try typing numbers in both fields and pressing enter each time.

Introducing a ScrollBar

Another way to improve the application would be to allow the user to input using a slider or scrollbar. Let's create one called `'TEMP.S'` ...

```
      SCR←'Scroll' ('Range' 101)('Event' 'Scroll' 'C2F')
      'TEMP.S' □WC SCR
```
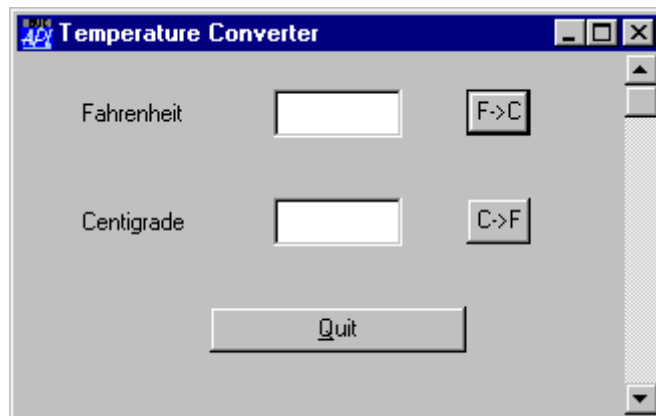
The range of a scrollbar goes from 1 to the value of the Range property. Setting Range to 101 will give us a range of 1-101. You will see in a moment why we need to do this. The Scroll event will be generated whenever the user moves the scrollbar. We have associated it with the callback function `'C2F'` which we will define as follows:

```
     ∇ C2F MSG
[1]   ⍝ Callback for Centigrade input via scrollbar
[2]     TEMP.C.Value←101-4⊃MSG
[3]     TEMP.F.Value←32+TEMP.C.Value×5÷9
     ∇
```

The Event message `MSG` contains information about the Scroll event. Its 4th element contains the requested thumb position. As we want to go from 0 at the top, to 100 at the bottom, we need to subtract this value from 101. This is done in line 2 of the function. `C2F[3]` calculates the corresponding Fahrenheit value.

Try moving the scrollbar and see what happens...
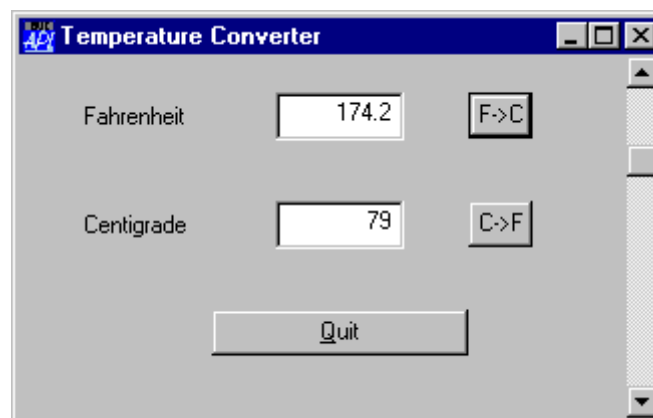
Adding a Menu

It would also be helpful if you could use the scrollbar to calculate in the reverse direction, from Fahrenheit to Centigrade. Let's add this facility, and give you the ability to choose to which scale the scrollbar applies through a menu.

To create a menu structure in a bar along the top of a Form (as opposed to a *floating* or *pop-up* menu) we first need to create a MenuBar object. This type of object has very few properties, and we need only specify its name, `'TEMP.MB'`.

```
'TEMP.MB' ⎕WC 'MenuBar'
```

Notice that, at this stage, there is no change in the appearance of the Form.
Then we can add a menu with the Caption `'Scale'`. The name of the menu is `'TEMP.MB.M'`. Adding the first menu causes the MenuBar to become visible.

```
'TEMP.MB.M' ⎕WC 'Menu' '&Scale'
```
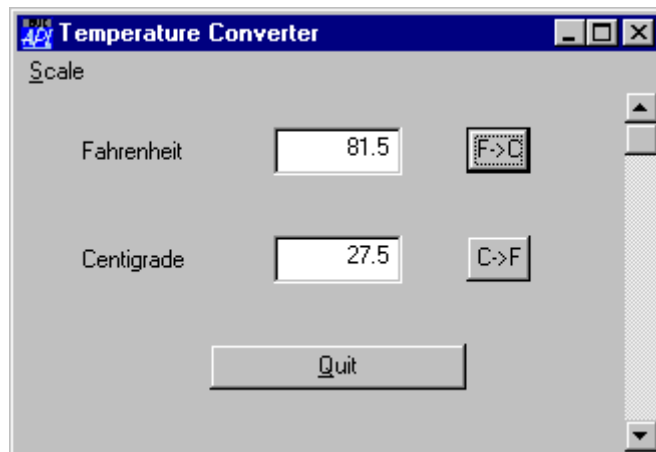

Note that the ampersand (&) allows the user to select the menu quickly by pressing "Alt+S".

Now we can add the two options to the menu. Note that the MenuBar and Menu objects do not represent final choices, they merely specify a path to a choice which is represented by the MenuItem object. When either of these is chosen, we want to execute a callback function that will make the necessary changes to the scrollbar. The statements to create each of these MenuItems are broken into 3 only to fit them onto the page.

First we create the Centigrade MenuItem...

```
C←'MenuItem' '&Centigrade'
C,←('Checked' 1)('Event' 'Select' 'SET_C')
'TEMP.MB.M.C' ⎕WC C
```

Setting the Checked property to 1 will cause a tick mark to appear against this option, indicating that it is the current one in force.

Then the Fahrenheit MenuItem...

```
F←'MenuItem' '&Fahrenheit'
F,←('Checked' 0)('Event' 'Select' 'SET_F')
'TEMP.MB.M.F' ⎕WC F
```

Notice that as the default value of Checked is 0, we didn't really have to set it explicitly for Fahrenheit. Nevertheless, it will do no harm to do so, and improves clarity.

The SET_C callback function is defined as follows:

```
     ∇ SET_C
[1]   ⍝ Sets the scrollbar to work in Centigrade
[2]    TEMP.S.Range'←101
[3]    TEMP.S.onScroll←'C2F'
[4]    TEMP.MB.M.C.Checked←1
[5]    TEMP.MB.M.F.Checked←0
     ∇
```

Line [2] simply sets the Range property of the scrollbar to be 101, and line [3] makes C2F the callback function when the scrollbar is moved. Lines [4] and [5] ensure that the tick mark is set on the chosen option.

The SET_F function is very similar...

```
      ∇ SET_F
[1]    ⍝ Sets the scrollbar to work in Fahrenheit
[2]     TEMP.S.Range←213
[3]     TEMP.S.onScroll←'F2C'
[4]     TEMP.MB.M.F.Checked ← 1
[5]     TEMP.MB.M.C.Checked ← 0
      ∇
```

and of course we need F2C to make the scrollbar work in Fahrenheit.

```
      ∇ F2C Msg;C;F
[1]    ⍝ Callback for Fahrenheit input via scrollbar
[2]     TEMP.F.Value←213-4⊃Msg
[3]     TEMP.C.Value←(TEMP.F.Value-32)×5÷9
      ∇
```

Running from Desktop

Now that we have a final working application, it would be nice to add it as a shortcut, so that the user can run it from the Start Menu or from the Desktop, like any other application.

First we need to define ⎕LX so that the application starts automatically.

```
⎕LX ← '⎕DQ''.'''
```

or, to avoid so many confusing quotes...

```
      ⎕LX ← ⎕
⎕DQ '.'
```

Next, it would be a good idea to clear the edit fields and ensure that the scrollbar is in its default position:

```
'TEMP.F' ⎕WS 'Text' ''
'TEMP.C' ⎕WS 'Text' ''
'TEMP.S' ⎕WS 'Thumb' 1
```

Then we must)SAVE the workspace...

```
      )SAVE TEMP
TEMP saved ...
```

... and exit APL

```
      )OFF
```

The next step is to add the application to the Desktop. This is done in the normal way, i.e.

Right-click on the Desktop and choose "New" followed by "Shortcut".

Type in the appropriate command line.
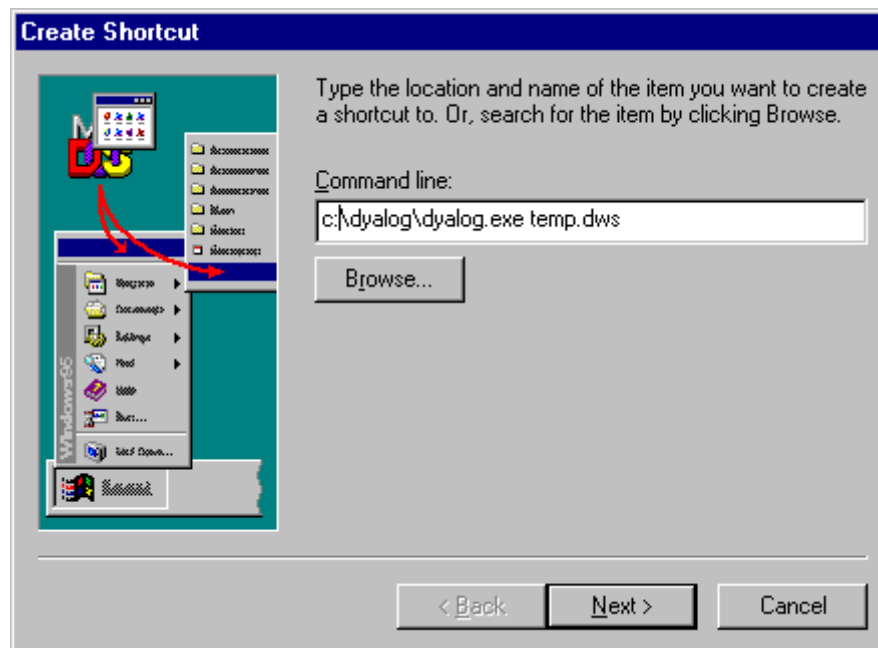
Select "Next" and give the application a name, then select "Finish".

The resulting icon is shown below. Note that although by default you will get a standard Dyalog APL icon, you could of course select another one from elsewhere on your system.

Temperature
Converter

Clicking on this icon will start your application. Notice that the APL Session Window will NOT appear at any stage unless there is an error in your code. All the user will see is your "Temperature Converter" dialog box.

Using ⎕NEW instead of ⎕WC

From Version 11 onwards, it is possible to use ⎕NEW to create Instances of the built-in GUI Classes. The following function illustrates this approach using the Temperature Converter example described previously.

```
      ∇ TempConv;TITLE;TEMP
[1]    TITLE←'Temperature Converter'
[2]    TEMP←⎕NEW'Form'((('Caption'TITLE)('Posn'(10 10))
                       ('Size'(30 40)))
[3]
[4]    TEMP.(MB←⎕NEW⊂'MenuBar')
[5]    TEMP.MB.(M←⎕NEW'Menu'(,⊂'Caption' '&Scale'))
[6]    TEMP.MB.M.(C←⎕NEW'MenuItem'
               (('Caption' '&Centigrade')('Checked' 1)))
[7]    TEMP.MB.M.(F←⎕NEW'MenuItem'
               (,⊂('Caption' '&Fahrenheit')))
[8]
[9]    TEMP.(LF←⎕NEW'Label'(('Caption' 'Fahrenheit')
                           ('Posn'(10 10))))
[10]   TEMP.(F←⎕NEW'Edit'(('Posn'(10 40))('Size'(θ 20))
                          ('FieldType' 'Numeric')))
[11]
[12]   TEMP.(LC←⎕NEW'Label'(('Caption' 'Centigrade')
                           ('Posn'(40 10))))
[13]   TEMP.(C←⎕NEW'Edit'(('Posn'(40 40))('Size'(θ 20))
                          ('FieldType' 'Numeric')))
[14]
[15]   TEMP.(F2C←⎕NEW'Button'(('Caption' 'F->C')
                             ('Posn'(10 70))('Default' 1)))
[16]   TEMP.(C2F←⎕NEW'Button'(('Caption' 'C->F')
                             ('Posn'(40 70))))
[17]   TEMP.(Q←⎕NEW'Button'(('Caption' '&Quit')
                           ('Posn'(70 30))('Size'(θ 40))
                           ('Cancel' 1)))
[18]
[19]   TEMP.(S←⎕NEW'Scroll'(⊂('Range' 101)))
[20]
[21]   TEMP.MB.M.C.onSelect←'SET_C'
[22]   TEMP.MB.M.F.onSelect←'SET_F'
[23]   TEMP.F2C.onSelect←'f2c'
[24]   TEMP.F.onGotFocus←'SET_DEF'
[25]   TEMP.C2F.onSelect←'c2f'
[26]   TEMP.C.onGotFocus←'SET_DEF'
[27]   TEMP.onClose←'QUIT'
[28]   TEMP.Q.onSelect←'QUIT'
[29]   TEMP.S.onScroll←'c2f_scroll'
[30]
[31]   ⎕DQ'TEMP'
      ∇
```

For brevity, only a couple of the callback functions are shown here.

```
      ∇ f2c
[1]     TEMP.C.Value←(TEMP.F.Value-32)×5÷9
      ∇

      ∇ c2f_scroll MSG
[1]    ⍝ Callback for Centigrade input via scrollbar
[2]     TEMP.C.Value←101-4⊃MSG
[3]     c2f
      ∇
```
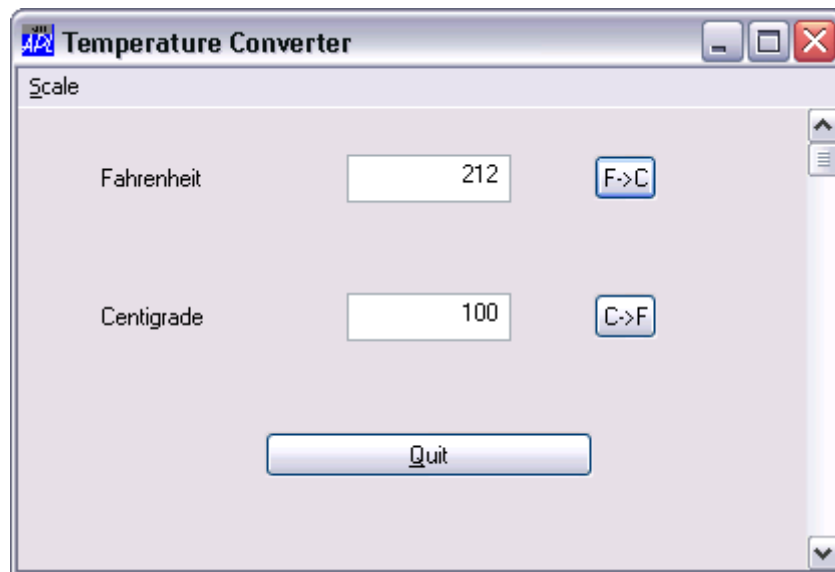
Temperature Converter Class

You may create user-defined Classes based upon Dyalog GUI objects as illustrated by the Temperature Converter Class which is listed overleaf.

To base a Class on a Dyalog GUI object, you specify the *name* of the object as its Base Class. For example, the Temperature Converter is based upon a Form:

```
:Class Temp: 'Form'
```

Being based upon a top-level GUI object, the Temperature Converter may be used as follows:

```
T1←□NEW Temp(⊂'Posn'(68 50))
```
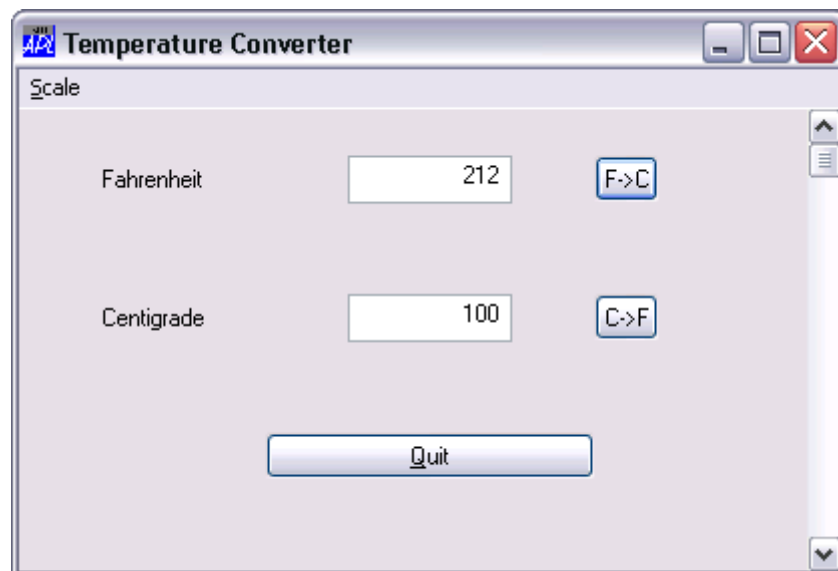
Temperature Converter Example

```
:Class Temp: 'Form'
    ∇ Make pv;TITLE
      :Access Public
      TITLE←'Temperature Converter'
      :Implements Constructor :Base (⊂'Caption' TITLE),pv,
                                    ⊂('Size' (30 40))
      MB←□NEW⊂'MenuBar'
      MB.(M←□NEW'Menu'(,⊂'Caption' '&Scale'))
      MB.M.(C←□NEW'MenuItem'(('Caption' '&Centigrade')
                              ('Checked' 1)))
      MB.M.(F←□NEW'MenuItem'(,⊂('Caption' '&Fahrenheit')))
      LF←□NEW'Label'(('Caption' 'Fahrenheit')
                     ('Posn'(10 10)))
      F←□NEW'Edit'(('Posn'(10 40))('Size'(⍬ 20))
                   ('FieldType' 'Numeric'))
      LC←□NEW'Label'(('Caption' 'Centigrade')
                     ('Posn'(40 10)))
      C←□NEW'Edit'(('Posn'(40 40))('Size'(⍬ 20))
                   ('FieldType' 'Numeric'))
      F2C←□NEW'Button'(('Caption' 'F->C')('Posn'(10 70))
                        ('Default' 1))
      C2F←□NEW'Button'(('Caption' 'C->F')('Posn'(40 70)))
      Q←□NEW'Button'(('Caption' '&Quit')('Posn'(70 30))
                      ('Size'(⍬ 40))('Cancel' 1))
      S←□NEW'Scroll'(⊂('Range' 101))
      MB.M.C.onSelect←'SET_C'
      MB.M.F.onSelect←'SET_F'
      F2C.onSelect←'f2c'
      F.onGotFocus←'SET_DEF'
      C2F.onSelect←'c2f'
      C.onGotFocus←'SET_DEF'
      onClose←'QUIT'
      Q.onSelect←'QUIT'
      S.onScroll←'c2f_scroll'
    ∇

    ∇ f2c
      C.Value←(F.Value-32)×5÷9
    ∇
    ∇ c2f
      F.Value←32+C.Value×9÷5
    ∇
    ∇ c2f_scroll MSG
      ⍝ Callback for Centigrade input via scrollbar
      C.Value←101-4⊃MSG
      c2f
    ∇
```

```
      ∇ f2c_scroll Msg
        ⍝ Callback for Fahrenheit input via scrollbar
        F.Value←213-4⊃Msg
        f2c
      ∇
      ∇ Quit
        Close
      ∇
      ∇ SET_DEF MSG
        (⊃MSG).Default←1
      ∇
      ∇ SET_C
        ⍝ Sets the scrollbar to work in Centigrade
        S.Range←101
        S.onScroll←'c2f_scroll'
        MB.M.C.Checked←1
        MB.M.F.Checked←0
      ∇
      ∇ SET_F
        ⍝ Sets the scrollbar to work in Fahrenheit
        S.Range←213
        S.onScroll←'f2c_scroll'
        MB.M.F.Checked←1
        MB.M.C.Checked←0
      ∇
:EndClass ⍝ Temp
```

Notice that the `:Implements Constructor` statement of its Constructor `Make`:

```
:Implements Constructor :Base (⊂'Caption' TITLE),pv,
                               ⊂('Size' (30 40))
```

passes on the application-specific property list (`pv`) given as its argument, but (in this case) specifies Caption and Size as well. The order in which the properties are specified in the `:Base` call ensures that the former will act as a default (and be overridden by an application-specific Caption requested in `pv`), whereas the specified Size of`(30 40)` will override whatever value of Size is requested by the host application in `pv`.

Other Instances can co-exist with the first:

```
        T2←⎕NEW Temp(('Caption' 'My Application')
                    ('Posn'(10 10))
```

Dual Class Example

The Dual Class example is based upon the example used to illustrate how you may build an ActiveX Control in Dyalog APL (see Chapter 13), but re-engineered as a internal Dyalog APL Class. The full listing of the Dual Class script is provided overleaf.

This version of Dual is based upon a SubForm:

```
:Class Dual: 'SubForm'
```

The Dual Control requires a GUI parent but several Instances can co-exist, quite independently, in the same parent.

For example, function `RUN` creates a Form and 3 Instances of Dual; one to convert Centigrade to Fahrenheit, one to convert Fahrenheit to Centigrade, and the third to convert centimetres to inches.

```
      ∇ RUN;F;D1PROPS;D2PROPS;D3PROPS
[1]
[2]    F←□NEW'Form'((('Caption' 'Dual Instances')
                  ('Coord' 'Pixel')('Size'(320 320)))
[3]
[4]    D1PROPS←('Caption1' 'Centigrade')
              ('Caption2' 'Fahrenheit')
[5]    D1PROPS,←('Intercept' 32)('Gradient'(9÷5))
              ('Value1' 0)('Range'(0 100))
[6]    F.D1←F.□NEW Dual((('Coord' 'Pixel')('Posn'(10 10))
                      ('Size'(100 300)),D1PROPS)
[7]
[8]    D2PROPS←('Caption1' 'Fahrenheit')
              ('Caption2' 'Centigrade')
[9]    D2PROPS,←('Intercept'(-32×5÷9))('Gradient'(5÷9))
              ('Value1' 0)('Range'(0 212))
[10]   F.D2←F.□NEW Dual((('Coord' 'Pixel')('Posn'(110 10))
                      ('Size'(100 300)),D2PROPS)
[11]
[12]   D3PROPS←('Caption1' 'Centimetres')
              ('Caption2' 'Inches')
[13]   D3PROPS,←('Intercept' 0)('Gradient'(÷2.54))
              ('Value1' 0)('Range'(0 100))
[14]   F.D3←F.□NEW Dual((('Coord' 'Pixel')('Posn'(210 10))
                      ('Size'(100 300)),D3PROPS)
[15]
[16]   □DQ'F'
      ∇
```
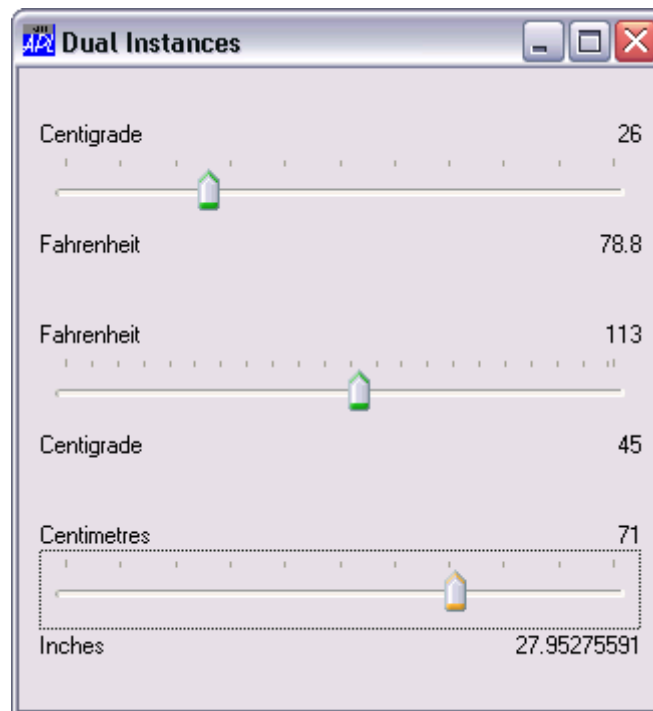
Dual's Constructor `Make` first splits its constructor arguments into those that apply to the Dual Class itself, and those that apply to the SubForm. Its `:Implements Constructor` statement then passes these on to the Base Constructor, together with an appropriate setting for EdgeStyle.

```
:Implements Constructor :Base BaseArgs,
                             ⊂'EdgeStyle' 'Dialog'
```

Dual Class Example

```
:Class Dual: 'SubForm'
    :Include GUITools
    :Field Private _Caption1←''
    :Field Private _Caption2←''
    :Field Private _Value1←0
    :Field Private _Value2←0
    :Field Private _Range←0
    :Field Private _Intercept←0
    :Field Private _Gradient←1
    :Field Private _Height←40

    ∇ Create args;H;W;POS;SH;CH;Y1;Y2;BaseArgs;MyArgs;
                  Defaults
      :Access Public
      MyArgs BaseArgs←SplitNV args
      :Implements Constructor :Base BaseArgs,
                                   ⊂'EdgeStyle' 'Dialog'
      ExecNV_ MyArgs ⍝ Set Flds named _PropertyName MyArgs
      Coord←'Pixel'
      H W←Size
      POS←2↑⌊0.5×0⌈(H-_Height)
      CH←⊃GetTextSize'W'
      'Slider'⎕WC'TrackBar'POS('Size'_Height W)
      Slider.(Limits AutoConf)←_Range 0
      Slider.(TickSpacing TickAlign)←10 'Top'
      Slider.onThumbDrag←'ChangeValue'
      Slider.onScroll←'ChangeValue'
      Y1←POS[1]-CH+1
      Y2←POS[1]+_Height+1
      'Caption1_'⎕WC'Text'_Caption1(Y1,0)('AutoConf' 0)
      'Caption2_'⎕WC'Text'_Caption2(Y2,0)('AutoConf' 0)
      'Value1_'⎕WC'Text'(⍕_Value1)(Y1,W)('HAlign' 2)
                      ('AutoConf' 0)
      CalcValue2
      'Value2_'⎕WC'Text'(⍕_Value2)(Y2,W)('HAlign' 2)
              ('AutoConf' 0)
      onConfigure←'Configure'
    ∇

    :Property Caption1, Caption2
    :Access Public
        ∇ R←Get arg
          R←(arg.Name,'_')⎕WG'Text'
        ∇
        ∇ Set arg
          (arg.Name,'_')⎕WS'Text'arg.NewValue
        ∇
    :EndProperty
```

```
:Property Value1
:Access Public
    ∇ R←Get
      R←_Value1
    ∇
    ∇ Set arg
      □NQ'Slider' 'Scroll' 0 arg.NewValue
    ∇
:EndProperty

:Property Intercept, Gradient, Range, Height, Value2
:Access Public
    ∇ R←Get arg
      R←⍕'_',arg.Name
    ∇
:EndProperty

∇ CalcValue2
  _Value2←_Intercept+_Gradient×_Value1
∇

∇ ChangeValue MSG
  ⍝ Callback for ThumbDrag and Scroll
  _Value1←⊃¯1↑MSG
  CalcValue2
  Value1_.Text←⍕_Value1
  Value2_.Text←⍕_Value2
∇

∇ Configure MSG;H;W;POS;CH;Y1;Y2
  2 □NQ MSG
  H W←Size
  POS←2↑⌊0.5×(H-_Height)
  CH←⊃GetTextSize'W'
  Slider.(Posn Size)←POS(_Height W)
  Y1←POS[1]-CH+1
  Y2←POS[1]+_Height+1
  Caption1_.Points←1 2ρY1,0
  Caption2_.Points←1 2ρY2,0
  Value1_.Points←1 2ρY1,W
  Value1_.Points←1 2ρY2,W
∇

:EndClass ⍝ Dual
```

CHAPTER 3

Graphics

Introduction

Graphical output is performed using the following objects:

Circle   Draws circles, arcs and pie-charts
Ellipse   Draws ellipses
Image   Displays or prints bitmaps and icons
Marker   Draws one or more sets of polymarkers
Poly   Draws one or more sets of polylines
Text   Displays or prints text
Rect   Draws rectangles

These graphical objects can be drawn in (i.e. be the children of) a wide range of other objects including a Form, Static, Printer and Bitmap.

Additional graphical resources are provided by the following objects. These are unusual in that they are not visible except when referenced as the property of another object:

Bitmap   Provides a Windows bitmap
Cursor   Provides a non-standard cursor
Icon   Provides a Windows icon
MetaFile   Provides a Windows Metafile

Graphical objects are created, like any other object, using `⎕WC` and have properties that can be changed using `⎕WS` and queried using `⎕WG`. Graphical objects also generate certain events.

Drawing Lines

To draw a line you use the Poly object. The following example draws a line in a Form from the point (y=10, x=5) to the point (y=90, x=60) :

```
'F'       ⎕WC 'Form' 'Drawing Lines' ('Size' 25 50)
'F.Line' ⎕WC 'Poly' ((10 90)(5 60))
```
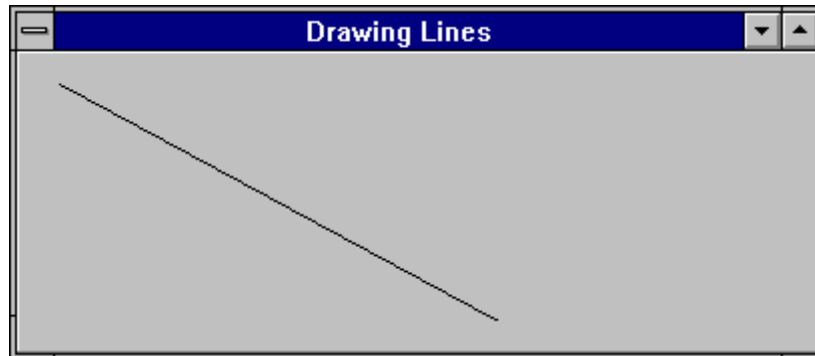

In the above example, the points are specified as a 2-element nested vector containing y-coordinates and x-coordinates respectively. You can also use a 2-column matrix, e.g.

```
'F.Line'⎕WC'Poly'(4 2ρ90 5 5 50 90 95 90 5)
```
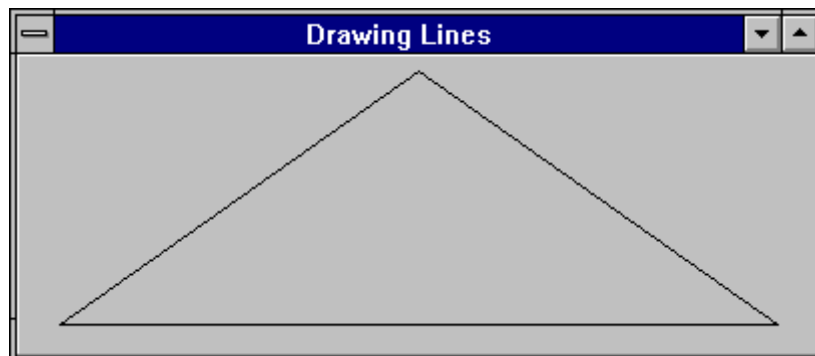

Notice that because the second example **replaced** the object F.Line, the original line drawn in the first example has been erased.

In common with the position and size of other GUI objects, y-coordinates precede x-coordinates. Graphical software typically uses (x,y) rather than (y,x) but the latter is consistent with the order in which coordinates are specified and reported for all other objects and events. The Dyalog APL GUI support allows you to freely mix graphical objects with other GUI components (for example, you can use the graphical Text object in place of a Label) and this (y,x) consistency serves to avoid confusion.

When a graphical object in a screen object is erased its parent is restored to the appearance that it had before that graphical object was created. Thus:

```
'F.Line' ⎕WC 'Poly' (2 2⍴10 5 50 10)
⎕EX 'F.Line'
```

first draws a line and then removes it. The following expression clears all graphical objects (and any other non-graphical ones too) from a parent object `'F'`:

```
⎕EX ⎕WN'F'
```

Similarly, objects automatically disappear when a function in which they are localised exits.

Erasing graphical objects that have been drawn on a Printer has no effect. Once drawn they cannot be undrawn.

Drawing in a Bitmap

A bitmap is an invisible resource (in effect, an area of memory) that is only displayed on the screen when it is referenced by another object. Any of the seven graphical objects (Circle, Ellipse, Image, Marker, Poly, Text and Rect) can be drawn in a bitmap (represented by a Bitmap object), using exactly the same ⎕WC syntax as if you were drawing in a Form, Static or Printer. However, drawing in a Bitmap is, like drawing on a Printer, an operation that cannot be "undone".

This facility allows you to construct a picture using lines, circles, text etc. and then later display it or save it as a bitmap.

Multiple Graphical Items

All graphical output objects (Circle, Ellipse, Image, Marker, Poly, Text and Rect) permit nested arguments so that you can draw several items with a single object. This feature has several advantages. Firstly, it allows you to treat related graphical items as a single object with a single name. This reduces the potential number of objects in existence and reduces the number of program statements needed to draw them. For example, sets of tick marks or grid lines do not have to be drawn separately, but can be represented by one object. Furthermore, because a set of lines can be embodied in a single object, you can erase them, replace them or drag/drop them as a unit. A further consideration is performance. A set of graphical items represented by a single object will normally be drawn faster than if each item was represented by separate objects.

For example, the following statements draw two separate rectangles; a red one at (y=10, x=20) and a blue one at (y=50, x=60). Both rectangles are size (30,30).

```
RED BLUE ← (255 0 0)(0 0 255)

'F.R1' ⎕WC 'Rect' (10 20)(30 30) ('FCol' RED)
'F.R2' ⎕WC 'Rect' (50 60)(30 30) ('FCol' BLUE)
```

The next statement achieves the same result, but uses only one object:

```
'F.R' ⎕WC 'Rect' ((10 50)(20 60)) (30 30)
                    ('FCol' RED BLUE)
```

The rectangles drawn by both these sets of statements are shown below (blue and red have been replaced by black for clarity).
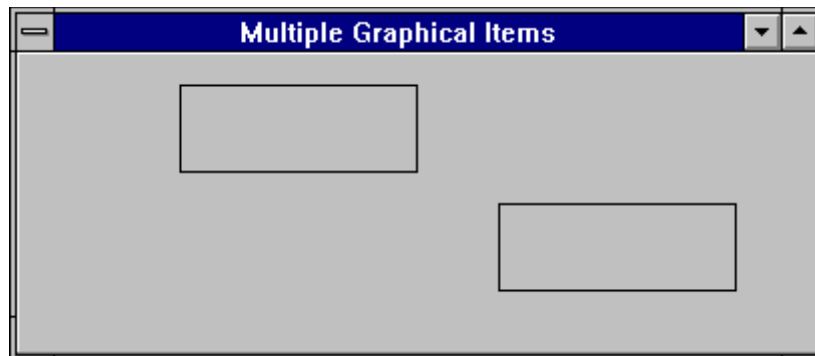
The capability to specify more than one graphical item as a single object is particularly useful with the Text object as it allows you to display or print several text items (at different positions and in different colours if you wish) in a single statement. For example, the following expressions display a set of "labels" in a Form `'F1'`:

```
LAB←'Name' 'Age' 'Address'
POS←3 2ρ10 10 10 60 30 10
'F1.LABS' ⎕WC 'Text' LAB POS
```
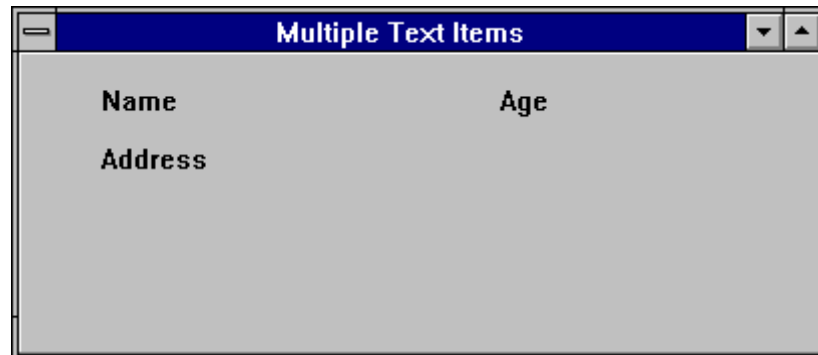
Unnamed Graphical Objects

When using the seven graphical output objects, you can optionally omit the final part of the name. For example, the following expression is valid:

```
'F.' ⎕WC 'Poly' (2 2⍴10 5 50 10)
```

When you create a **named** object, all of the properties pertaining to that object are stored internally in your workspace. A polyline consisting of a large number of points thus takes up a significant amount of memory. However, this is necessary because the APL interpreter needs the information in order to redraw the object when another window is placed over it and then moved away again (exposure) or when the user resizes the Form in which it is displayed.

When you create an **unnamed** graphical object, the object is drawn, but its properties are **not** remembered internally, thus conserving workspace. This has two consequences. Firstly, you cannot subsequently modify or query the object's properties; you must name an object if you are ever going to refer to it again. Secondly, the object cannot automatically be redrawn (by APL) when it is exposed or resized. Instead, you must control this yourself using the Expose event.

Unnamed graphical objects are useful in the following circumstances:

- For output to a Printer.

- When you are very short of workspace.

- When you are sure that the window you are drawing in will not need to be redrawn, for example, when you are working "full-screen".

- For drawing in a Bitmap or a Metafile.

- For creating bitmaps in an ImageList

Bitmaps and Icons

Bitmaps and icons are implemented as separate objects that you can create and destroy. Once you have created such an object you can reference it as many times as you wish. For example, you can use the same bitmap in several Buttons or associate the same icon with several Forms.

The Bitmap and Icon objects can be created in one of two ways. They are either loaded from an existing file or they are defined from APL arrays.

The files concerned must be in the appropriate Windows format for the object (.BMP or .ICO files) which can be edited by a standard Windows utility such as Paintbrush. The following example creates a Bitmap object from the CARS.BMP bitmap file which is supplied in the WS sub-directory:

```
ROOT←'C:\Program Files\Dyalog\Dyalog APL 12.1 Unicode\'
'CARS' ⎕WC 'Bitmap' (ROOT,'\WS\CARS')
```

Then you can use the Bitmap to fill the background of a Form by:

```
'F1' ⎕WC 'Form' ('Picture' 'CARS' 1)('Size' 25 50)
```


The "1" in the expression specifies that the Bitmap is to be used to "tile" the background of the Form. The result is shown in the illustration below. You can also position the Bitmap in the top-left (0) or centre (3) of the Form, or even have the Bitmap scaled automatically (2) to fit exactly in the Form. These settings are useful for displaying pictures. You can explore these facilities using the BMVIEW function in the UTIL workspace.

Instead of creating Bitmap and Icon objects from file, you can define them using APL arrays. These arrays specify the individual pixels that make up the picture or shape of the object in question.

There are two ways to define a Bitmap object from APL arrays. The first method, which is limited to colour palettes of 16 or 256 colours is to supply two arrays; one containing the colour indices for every pixel in the bitmap, and one containing the colour map. The colour map specifies the colours (in terms of their red, green and blue components) corresponding to the indices in the first array. For example, the following expressions create a 32 x 32 Bitmap from the arrays `PIX` and `CM`:

```
      ρPIX  ⍝ colour index (in CM) of each pixel
32 32
      ρCM   ⍝ 16-row matrix of RGB values
16 3
      'BM' ⎕WC 'Bitmap' ('Bits' PIX)('CMap' CM)
```

The reason that this method is restricted to 256 colours is that the CMap array containing the colour map is, of necessity, the same size as the colour palette. Even for a relatively modest 16-bit colour palette, the size of the array would be 65536 x 3.

The second method, which applies to all sizes of colour palette, is to use a single array that represents each pixel by a number that is an encoding of the red, green and blue components. The formula used to calculate each pixel value is:

```
      256⊥RED GREEN BLUE
```

where `RED`, `GREEN` and `BLUE` are integers in the range 0-255.

Thus the example above can be achieved using a single array `CBITS` as follows:

```
      CBITS←(256⊥⍉CMAP)[⎕IO+PIX]
      'BM' ⎕WC 'Bitmap' ('CBits' CBITS)
```

You can build APL arrays representing bitmaps using the `BMED` function in the `BMED` workspace. You can also load them from file, e.g.

```
      'BM' ⎕WC 'Bitmap' 'C:\WINDOWS\CARS'
      PIX CM ← 'BM' ⎕WG 'Bits' 'CMap'
```

Metafiles

A Windows metafile is a mechanism for representing a picture as a collection of graphics commands. Once a metafile has been created, the picture that it represents can be drawn repeatedly from it. Metafiles are device-independent, so the picture can be reproduced on different devices. Unlike bitmaps, metafiles can be scaled accurately and are therefore particularly useful for passing graphical information between different applications. Note that some other applications only support *placeable* metafiles. See RealSize property for details.

Creating a Metafile Object

In Dyalog APL, a Windows metafile is represented by the Metafile object. This is created in much the same way as a Bitmap object. That is, you can either make a Metafile object from an existing .WMF file, or you can create an empty one and then draw onto it using Poly, Text and other graphical objects. For example, to create a Metafile object called BFLY from the BUTTRFLY.WMF metafile that comes with Microsoft Office, you can execute the following:

```
'BFLY' ⎕WC 'Metafile' 'c:\msoffice\clipart\buttrfly'
```

If instead you wanted to create a metafile drawing from scratch, you could do so as follows. Notice that there is no need to assign names to the graphical objects drawn onto the Metafile.

```
'METADUCK'  ⎕WC 'Metafile' ''
'METADUCK.' ⎕WC 'Poly' DUCK
'METADUCK.' ⎕WC 'Text' 'Quack' (25 86)
```

Drawing a Metafile Object

A Metafile object is drawn by specifying its name as the Picture property of another object. This causes the Metafile to be drawn in that object and scaled to fit exactly within its boundaries. The following statement causes the Metafile object called METADUCK to be drawn in the Form F1.

```
'F1'⎕WC'Form' ('Size' 25 50) ('Picture' 'METADUCK')
```
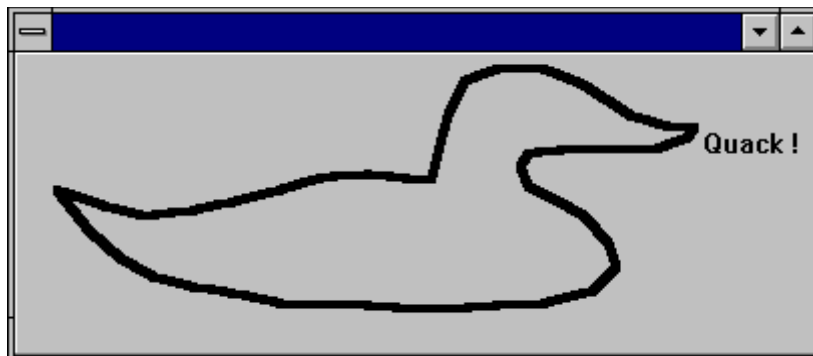
Picture Buttons

Picture buttons in *toolbars* are most conveniently represented by ToolButtons in
ToolControls (see Chapter 4). Pictures on stand-alone buttons or buttons used in the
(superseded) ToolBar object, may be created using Bitmap, Icon and Metafile objects
and there are two different methods provided. The first (and the simplest) is to use the
Picture property which applies to all 3 types of image,(Bitmap, Icon or Metafile). The
second method is to use the BtnPix property. This requires rather more effort, and only
draws Bitmaps, and not Icons or Metafiles. However, the BtnPix property gives you
total control over the appearance of a Button which the Picture property does not.

Using the Picture Property

The Picture property overlays a Bitmap, Icon or Metafile on top of a standard
pushbutton. The following examples use icon files distributed with Visual Basic. If you
don't have this product, you can use any other suitable icons, bitmaps or metafiles that
you have on your PC. Alternatively, you can create bitmaps using the `BMED` function in
the `BMED` workspace.

Load a Visual Basic icon:

```
'ARROW' ⎕WC 'Icon' 'C:\VB\IconS\ARROWS\ARW02DN'

'F' ⎕WC 'Form' 'Using the Picture Property'

'F.B' ⎕WC 'Button' ('Coord' 'Pixel')
                    ('Size' 40 40)
                    ('Picture' 'ARROW' 3)
```
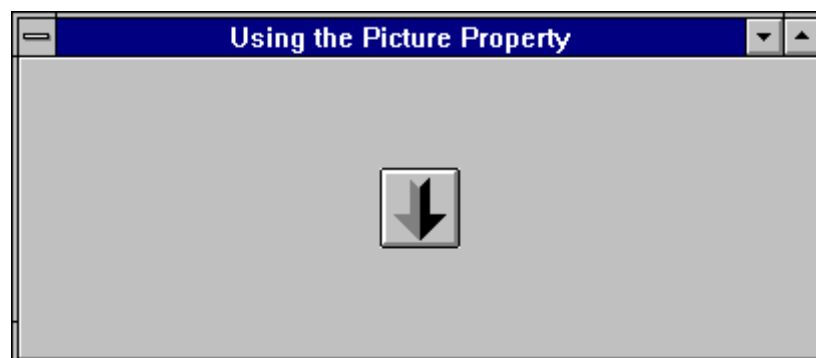
Notice that (by definition) an icon is 32 x 32 pixels in size. To allow space for the pushbutton borders you have to make the Button at least 40 x 40 pixels. The "3" means put the `'ARROW'` in the **centre** of the button.

When you press a Button which has its Picture property set like this, APL/W automatically shifts the overlaid image down and to the right by 1 pixel. This complements the change in appearance of the button borders and achieves a "pressed-in" look. When you release the button, APL shifts the image back again.

The Picture property therefore provides a very simple mechanism for implementing a "tool-button", especially if you already have a bitmap or icon file that you want to use.

However, the Picture property has certain limitations. Firstly, you cannot alter the "pressed-in" look of the Button which is determined automatically for you. You might want the Button to change colour when you press it, and you cannot achieve this with the Picture property. Secondly, the appearance of the Button is unchanged when you make it inactive (by setting its Active property to 0).

Note that if you use the Picture property on Radio or Check buttons, the buttons assume pushbutton appearance although their radio/check behaviour is unaffected.

Using the BtnPix Property

You can obtain **complete** control over the appearance of a Button by using the BtnPix property; however this entails more work on your part.

BtnPix allows you to associate three bitmaps with a Button, i.e.

> one for when the Button is in its normal state
> one for when it is pressed/selected
> one for when it is inactive

For example, if you have created three Bitmap objects called `'UP'`, `'DOWN'` and `'DEAD'`, you define the Button as follows:

```
'F.B' ⎕WC 'Button' ('BtnPix' 'UP' 'DOWN' 'DEAD')
```

APL subsequently displays one of the three Bitmap objects according to the state of the Button; i.e. `'UP'` for its normal state (State 0), `'DOWN'` for its pressed/selected state (State 1) or `'DEAD'` when it is inactive (Active 0).

The BtnPix property requires that you use Bitmap objects; it doesn't work for Icons. This is because icons are normally at least partly **transparent**. However, it is very easy to convert an icon file to a Bitmap object. First you create an Icon object from the icon (.ICO) file. Next you read the icon's pattern definition (Bits property) and colour map (CMap property) into the workspace. Then finally, you create a Bitmap from these two arrays.

The following example illustrates how you can make a Button from three Visual Basic icon files. You can also make your own trio of bitmaps using the `BTNED` function in the `BMED` workspace.

Load a Visual Basic icon:

```
'T1' ⎕WC 'Icon' 'C:\VB\IconS\TRAFFIC\TRFFC10A'
```

Read its Bits (pattern) and CMap (colour map):

```
Bits CMap ← 'T1' ⎕WG 'Bits' 'CMap'
```

Now define a Bitmap from these variables, (replacing the `T1` object):

```
'T1' ⎕WC 'Bitmap' '' Bits CMap
```

Now make a second Bitmap:

```
'T2' ⎕WC 'Icon' 'C:\VB\IconS\TRAFFIC\TRFFC10B'
'T2' ⎕WC 'Bitmap' '' , 'T2' ⎕WG 'Bits' 'CMap'
```

and a third:

```
'T3' ⎕WC 'Icon' 'C:\VB\IconS\TRAFFIC\TRFFC10C'
'T3' ⎕WC 'Bitmap' '' , 'T3' ⎕WG 'Bits' 'CMap'
```

Now define the Button:

```
'F.B' ⎕WC 'Button' ('BtnPix' 'T1' 'T2' 'T3')
```

What does it look like when inactive?

```
'F.B' ⎕WS 'Active' 0
```

Using Icons

You have seen how icons can be displayed using the Picture property. Other uses of icons are described below.

Firstly, you can associate an icon with a Form or so that the icon is displayed (by Windows) when the Form is minimised. This is done using the IconObj property. For example, the following expressions would associate the UK Flag icon distributed with Visual Basic with the Form `'F1'`. This icon would then be displayed when `'F1'` is minimised.

```
'FLGUK' ⎕WC 'Icon' 'C:\VB\ICONS\FLAGS\FLGUK'
'F1' ⎕WC 'Form' ('IconObj' 'FLGUK')
```

The IconObj property also applies to the Root object `'.'`. This defines the icon to be displayed for your **application** as a whole when the user toggles between applications using Alt+Tab. It is used in conjunction with the Caption property which determines the description of your application that is shown alongside the icon, e.g.

```
'MYIcon' ⎕WC 'Icon' ...
'.' ⎕WS ('IconObj' 'MYIcon') ('Caption' 'My System')
```

An icon can be displayed using the Image object. This object is used to position one or more Icon objects (or Bitmap objects) in a Form or Static. It can also be used to draw an icon on a Printer. If you make the Image dragable, you will be able to drag/drop the icon. The following example displays a dragable Icon at (10,10) in a Form. It also associates the callback function `'Drop'` with the DragDrop event so that this function is called when the user drag/drops the icon.

```
'F1' ⎕WC 'Form' ('Event' 'DragDrop' 'Drop')
'FLGUK' ⎕WC 'Icon' 'C:\VB\ICONS\FLAGS\FLGUK'
'F1.I' ⎕WC 'Image' (10 10) ('Picture' 'FLGUK')
                                ('Dragable' 2)
```
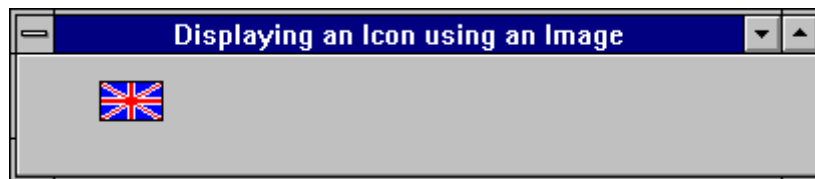

Notice that `('Dragable' 2)` specifies that an object is fully displayed while it is being dragged. The setting `('Dragable' 1)` causes only the bounding rectangle around the object to be dragged.

CHAPTER 4

Composite Controls

This chapter describes how to use the ToolControl, CoolBar, TabControl and StatusBar objects.

Several of these objects require the Windows Custom Control Library COMCTL32.DLL, Version 4.72 or higher.

The ToolControl and ToolButton Objects

The ToolControl object is normally used in conjunction with ToolButtons, although it may also act as a parent for other objects, including a MenuBar.

A ToolButton may display a Caption and an Image, although both are optional. Images for individual ToolButtons are not defined one-by-one, but instead are defined by an ImageList which contains a set of bitmaps or icons.

The ImageListObj property of the ToolControl specifies the name of one or more ImageList objects to be used. The ImageIndex properties of each of the ToolButtons specifies which of the images in each ImageList object apply to which of the ToolButtons.

Standard Bitmap Resources

Typically, you will want your ToolControls to provide standard Windows buttons and the easiest way to achieve this is to utilise the standard Windows bitmaps that are contained in COMCTL32.DLL. There are three main sets of bitmaps, each of which is provided in two sizes, small (16x16) and large (24 x 24).

Resource number 120 (IDB_STD_SMALL_COLOR) and 121 (IDB_STD_LARGE_COLOR) contain the following set of assorted bitmap images.

Resource number 124 (IDB_VIEW_SMALL_COLOR) and 125 (IDB_VIEW_LARGE_COLOR) contain a set of bitmaps relating to different views of information. These are used, for example in the Windows Explorer tool bar

Resource number 130 (IDB_HIST_SMALL_COLOR) and 131 (IDB_HIST_LARGE_COLOR) contain another useful set of bitmaps
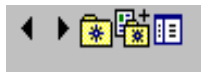
COMCTL32.DLL also contains individual bitmaps in resources 132-134.

Dyalog Bitmap Resources

Another three sets of useful bitmaps are to be found in the DYARES32.DLL file. These bitmaps are used in the Dyalog APL/W Session tool buttons. Note that if you include these bitmaps in a run-time application, you will have to ship DYARES32.DLL with it.

The *normal* set of bitmaps associated with the Session buttons may be created using the statement:

```
'bm'⎕wc'Bitmap' ('DYARES32' 'tb_normal')
```
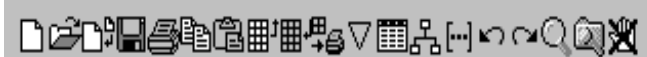
The bitmaps used when the buttons are *highlighted* may be created using the statement (note that the file name may be elided)

```
'bm'⎕wc'Bitmap' ('' 'tb_hot')
```
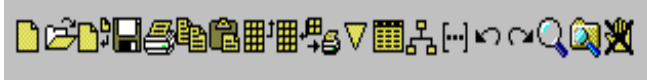
The bitmaps used when the buttons are *inactive* may be created using the statement

```
'bm'⎕wc'Bitmap' ('' 'tb_inactive')
```

Creating ImageLists for ToolButtons

You may use up to three ImageList objects to represent ToolButton images. These will be used to specify the pictures of the ToolButton objects in their normal, highlighted (sometimes termed hot) and inactive states respectively.

The set of images in each ImageList is then defined by creating unnamed Bitmap or Icon objects as children.

When creating an ImageList, it is a good idea to set its MapCols property to 1. This means that standard button colours used in the bitmaps will automatically be adjusted to take the user's colour preferences into account.

When you create each ToolButton you specify its ImageIndex property, selecting up to three pictures (normal, highlighted and inactive) to be displayed on the button.

If you specify only a single ImageList, the picture on the ToolButton will be the same in all three cases. However, the appearance of the buttons themselves change when the button is highlighted or pressed, and in many situations this may be sufficient behaviour.

The following example illustrates how a simple ToolControl can be constructed using standard Windows bitmaps. Notice that the Masked property of the ImageList is set to 0; this is necessary if the ImageList is to contain bitmaps, as opposed to icons. Secondly, because the bitmaps are in this case size 16 x 16, it is unnecessary to specify the Size property of the ImageList which is, by default, also 16 x 16.

```
'F'⎕WC'Form' 'ToolControl'('Size' 10 40)
'F.TB'⎕WC'ToolControl'
'F.TB.IL'⎕WC'ImageList'('Masked' 0) ('MapCols' 1)
'F.TB.IL.'⎕WC'Bitmap'('ComCtl32' 120)⍝ STD_SMALL
'F.TB'⎕WS'ImageListObj' 'F.TB.IL'
'F.TB.B1'⎕WC'ToolButton' 'New'('ImageIndex' 7)
'F.TB.B2'⎕WC'ToolButton' 'Open'('ImageIndex' 8)
'F.TB.B3'⎕WC'ToolButton' 'Save'('ImageIndex' 9)
```

The Style Property

The overall appearance of the ToolButton objects displayed by the ToolControl is defined by the Style property of the ToolControl itself, rather than by properties of individual ToolButtons.

Note that the Style property may only be set when the ToolControl is created using ⎕WC and may not subsequently be changed using ⎕WS.

Style may be `'FlatButtons'`, `'Buttons'`, `'List'` or `'FlatList'`. The default Style is of a ToolControl is `'FlatButtons'`, as is the first example above. The following examples illustrate the other three styles:
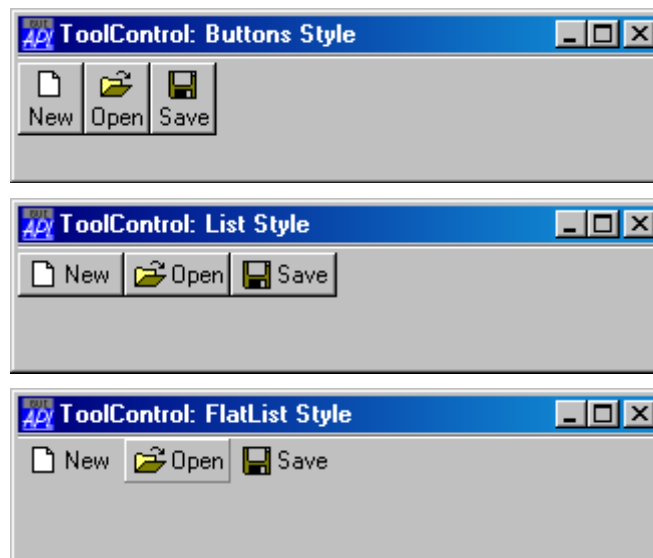
The Divider Property

You will notice that, in the above examples, there is a thin groove drawn above the ToolControl. The presence or absence of this groove is controlled by the Divider property whose default is 1. The following picture illustrates the effect of setting Divider to 0.
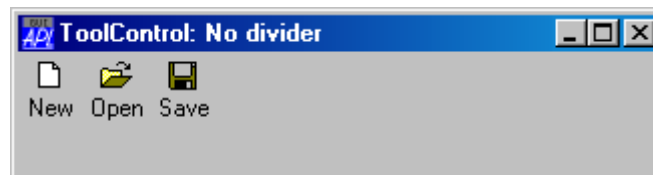
The MultiLine Property

The MultiLine property specifies whether or not ToolButtons (and other child controls) are arranged in several rows (or columns) when there are more than would otherwise fit.

If MultiLine is 0 (the default), the ToolControl object *clips* its children and the user must resize the Form to bring more objects into view.

Note that you may change MultiLine dynamically, using ⎕WS.

```
'F'⎕WC'Form' 'ToolControl: MultiLine 0'
'F.TB'⎕WC'ToolControl'('Style' 'List')

'F.TB.IL'⎕WC'ImageList'('Masked' 0)('Size' 24 24)
'F.TB.IL.'⎕WC'Bitmap'('ComCtl32' 121)⍝ STD_LARGE
'F.TB'⎕WS'ImageListObj' 'F.TB.IL'

'F.TB.B1'⎕WC'ToolButton' 'Cut'('ImageIndex' 1)
'F.TB.B2'⎕WC'ToolButton' 'Copy'('ImageIndex' 2)
'F.TB.B3'⎕WC'ToolButton' 'Paste'('ImageIndex' 3)
'F.TB.B4'⎕WC'ToolButton' 'Undo'('ImageIndex' 4)
'F.TB.B5'⎕WC'ToolButton' 'Redo'('ImageIndex' 5)
'F.TB.B6'⎕WC'ToolButton' 'Delete'('ImageIndex' 6)
```
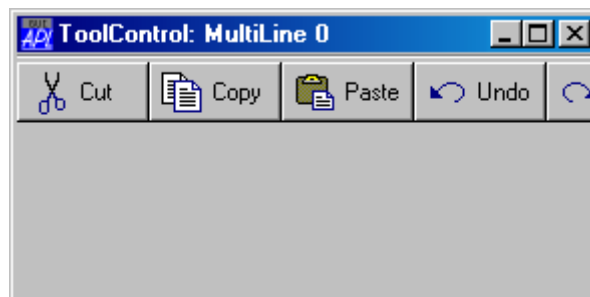

If we set MultiLine to 1, the ToolButtons are instead displayed in several rows:
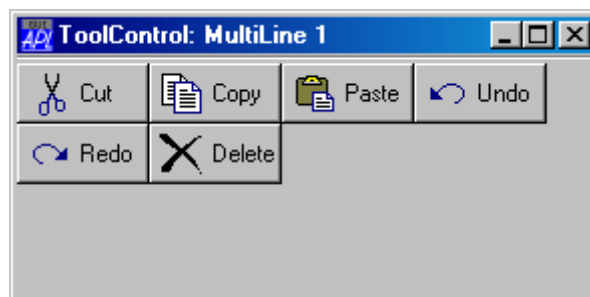
The Transparent Property

The Transparent property (default 0) specifies whether or not the ToolControl is transparent. Note that Transparent must be set when the object is created using ⎕WC and may not subsequently be changed using ⎕WS.

If a ToolControl is created with Transparent set to 1, the visual effect is as if the ToolButtons (and other controls) were drawn directly on the parent Form as shown below.

```
'F'⎕WC'Form' 'ToolControl: Transparent 1)'
'F.BM'⎕WC'Bitmap' 'C:\WINDOWS\WINLOGO'
'F'⎕WS'Picture' 'F.BM' 1

'F.TB'⎕WC'ToolControl'('Style' 'FlatList')('Transparent' 1)
'F.TB.IL'⎕WC'ImageList'('Masked' 0)('Size' 24 24)
'F.TB.IL.'⎕WC'Bitmap'('ComCtl32' 121)⍺ STD_LARGE
'F.TB'⎕WS'ImageListObj' 'F.TB.IL'

'F.TB.B1'⎕WC'ToolButton' 'New'('ImageIndex' 7)
'F.TB.B2'⎕WC'ToolButton' 'Open'('ImageIndex' 8)
'F.TB.B3'⎕WC'ToolButton' 'Save'('ImageIndex' 9)
```

Radio buttons, Check buttons and Separators

The Style property of a ToolButton may be `'Push'`, `'Check'`, `'Radio'`, `'Separator'` or `'DropDown'`.

Push buttons (the default) are used to generate actions and pop in and out when clicked.

Radio and *Check* buttons are used to select options and have two states, normal (out) and selected (in). Their State property is 0 when the button is in its normal (unselected state) or 1 when it is selected.

A group of adjacent ToolButtons with Style `'Radio'` defines a set in which only one of the ToolButtons may be selected at any one time. The act of selecting one will automatically deselect any other. Note that a group of Radio buttons must be separated from Check buttons or other groups of Radio buttons by ToolButtons of another Style.

Separator buttons are a special case as they have no Caption or picture, but appear as a thin vertical grooves that are used only to separate groups of buttons.

The following example illustrates how two groups of radio buttons are established by inserting a ToolButton with Style `'Separator'` between them. This ToolControl could be used to control the appearance of a ListView object. The first group is used to select the view (Large Icon, Small Icon, List or Report), and the second is used to sort the items by Name, Size or Date. In the picture, the user has selected Small Icon View and Sort by Date.

Notice that the appearance of the Separator ToolButton is less obvious when the ToolControl Style is Buttons or List, but the radio grouping effect is the same:

```
'F'⎕WC'Form' 'ToolControl: ToolButton Styles'('Size' 10 40)
'F.TB'⎕WC'ToolControl'

'F.TB.IL'⎕WC'ImageList'('Masked' 0)
'F.TB.IL.'⎕WC'Bitmap'('ComCtl32' 124)⍝ VIEW_SMALL
'F.TB'⎕WS'ImageListObj' 'F.TB.IL'

'F.TB.B1'⎕WC'ToolButton' 'Large'  ('ImageIndex' 1)
                                  ('Style' 'Radio')
'F.TB.B2'⎕WC'ToolButton' 'Small'  ('ImageIndex' 2)
                                  ('Style' 'Radio')
'F.TB.B3'⎕WC'ToolButton' 'List'   ('ImageIndex' 3)
                                  ('Style' 'Radio')
'F.TB.B4'⎕WC'ToolButton' 'Details'('ImageIndex' 4)
                                  ('Style' 'Radio')

'F.TB.S1'⎕WC'ToolButton'('Style' 'Separator')

'F.TB.B5'⎕WC'ToolButton' 'Name'   ('ImageIndex' 5)
                                  ('Style' 'Radio')
'F.TB.B6'⎕WC'ToolButton' 'Size'   ('ImageIndex' 6)
                                  ('Style' 'Radio')
'F.TB.B7'⎕WC'ToolButton' 'Date'   ('ImageIndex' 7)
                                  ('Style' 'Radio')
```

Drop-Down buttons

It is possible to define ToolButtons that display a drop-down menu from which the user may choose an option. This is done by creating a ToolButton with Style `'DropDown'`.

A ToolButton with Style `'DropDown'` has an associated popup Menu object which is named by its Popup property. There are two cases to consider.

If the ShowDropDown property of the parent ToolControl is 0, clicking the ToolButton causes the popup menu to appear. In this case, the ToolButton itself does not itself generate a Select event; you must rely on the user selecting a MenuItem to specify a particular action.

If the ShowDropDown property of the parent ToolControl is 1, clicking the dropdown button causes the popup menu to appear; clicking the ToolButton itself generates a Select event, but does not display the popup menu.

```
'F'⎕WC'Form' 'ToolControl: Dropdown Buttons' ('Size' 20 40)
'F.TB'⎕WC'ToolControl'('ShowDropDown' 1)

:With 'F.FMENU'⎕WC'Menu' ⍝ Popup File menu
    'NEW'⎕WC'MenuItem' '&New'
    'OPEN'⎕WC'MenuItem' '&Open'
    'CLOSE'⎕WC'MenuItem' '&Close'
:EndWith

:With 'F.EMENU'⎕WC'Menu' ⍝ Popup Edit menu
    'CUT'⎕WC'MenuItem' 'Cu&t'
    'COPY'⎕WC'MenuItem' '&Copy'
    'PASTE'⎕WC'MenuItem' '&Paste'
:EndWith

'F.TB.B1'⎕WC'ToolButton' 'File' ('Style' 'DropDown')
                                ('Popup' 'F.FMENU')
'F.TB.B2'⎕WC'ToolButton' 'Edit' ('Style' 'DropDown')
                                ('Popup' 'F.EMENU')
```
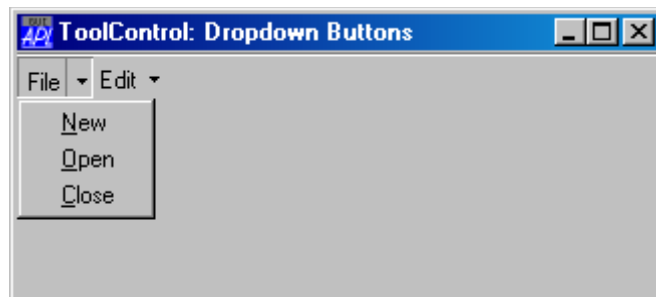
A MenuBar as the child of a ToolControl

As a special case, the ToolControl may contain a MenuBar as its **only** child. In this case, Dyalog APL/W causes the menu items to be drawn as buttons, even under Windows 95.

Although nothing is done to prevent it, the use of other objects in a ToolControl containing a MenuBar, is not supported.

```
'F'⎕WC'Form' 'ToolControl with MenuBar'('Size' 20 40)
'F.TB'⎕WC'ToolControl'

:With 'F.TB.MB'⎕WC'MenuBar'
    :With 'File'⎕WC'Menu' 'File'
        'New'⎕WC'MenuItem' 'New'
        'Open'⎕WC'MenuItem' 'Open'
        'Close'⎕WC'MenuItem' 'Close'
    :EndWith

    :With 'Edit'⎕WC'Menu' 'Edit'
        'Cut'⎕WC'MenuItem' 'Cut'
        'Copy'⎕WC'MenuItem' 'Copy'
        'Paste'⎕WC'MenuItem' 'Paste'
    :EndWith

:EndWith
```
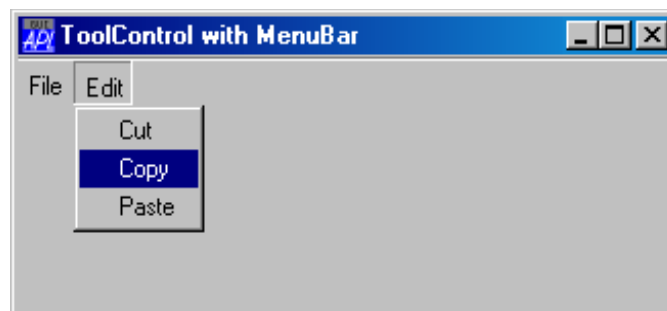
Providing User Customisation

It is common to allow the user to switch the appearance of a ToolControl dynamically. This may be done using a pop-up menu. In addition to providing a choice of styles, the user may switch the text captions on and off.

The ShowCaptions property specifies whether or not the captions of ToolButton objects are drawn. Its default value is 1 (draw captions).

ToolButtons drawn without captions occupy much less space and ShowCaptions provides a quick way to turn captions on/off for user customisation.
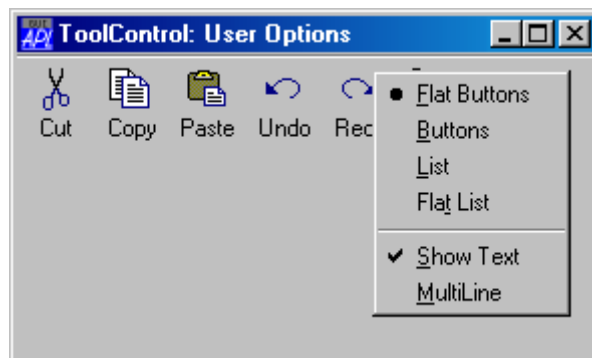
The following functions illustrate how this was achieved.

```
     ∇ Example
[1]    'F'⎕WC'Form' 'ToolControl: User Options'
[2]    'F.TB'⎕WC'ToolControl'
[3]    'F.TB'⎕WS'Event' 'MouseDown' 'TC_POPUP'
[4]
[5]    'F.TB.IL'⎕WC'ImageList'('Masked' 0)('Size' 24 24)
[6]    'F.TB.IL.'⎕WC'Bitmap'('ComCtl32' 121)⍝ STD_LARGE
[7]    'F.TB'⎕WS'ImageListObj' 'F.TB.IL'
[8]
[9]    'F.TB.B1'⎕WC'ToolButton' 'Cut'('ImageIndex' 1)
[10]   'F.TB.B2'⎕WC'ToolButton' 'Copy'('ImageIndex' 2)
[11]   'F.TB.B3'⎕WC'ToolButton' 'Paste'('ImageIndex' 3)
[12]   'F.TB.B4'⎕WC'ToolButton' 'Undo'('ImageIndex' 4)
[13]   'F.TB.B5'⎕WC'ToolButton' 'Redo'('ImageIndex' 5)
[14]   'F.TB.B6'⎕WC'ToolButton' 'Delete'('ImageIndex' 6)
     ∇
```

```
      ∇ TC_POPUP MSG;popup;TC;STYLE;SHOW;MULTI;OPTION
[1]     ⍝ Popup menu on ToolControl
[2]     :If (2≠5⊃MSG) ⍝ Right mouse button ?
[3]         :Return
[4]     :EndIf
[5]
[6]     TC←'#.',⊃MSG
[7]     STYLE SHOW MULTI←TC ⎕WG'Style' 'ShowCaptions'
                              'MultiLine'
[8]
[9]     :With 'popup'⎕WC'Menu'
[10]        'FlatButtons'⎕WC'MenuItem' '&Flat Buttons'
                          ('Style' 'Radio')
[11]        'Buttons'⎕WC'MenuItem' '&Buttons'
                          ('Style' 'Radio')
[12]        'List'⎕WC'MenuItem' '&List'('Style' 'Radio')
[13]        'FlatList'⎕WC'MenuItem' 'Fla&t List'
                          ('Style' 'Radio')
[14]        STYLE ⎕WS'Checked' 1
[15]        'sep'⎕WC'Separator'
[16]        'ShowCaptions'⎕WC'MenuItem' '&Show Text'
                          ('Checked'SHOW)
[17]        'MultiLine'⎕WC'MenuItem' '&MultiLine'
                          ('Checked'MULTI)
[18]
[19]        ('MenuItem'⎕WN'')⎕WS¨⊂'Event' 'Select' 1
[20]
[21]        :If 0=⍴MSG←⎕DQ''
[22]            :Return
[23]        :EndIf
[24]
[25]        :Select OPTION←⊃MSG
[26]        :CaseList 'FlatButtons' 'Buttons' 'List'
                      'FlatList'
[27]            TC ⎕WS'Style'OPTION
[28]        :Else
[29]            TC ⎕WS OPTION(~TC ⎕WG OPTION)
[30]        :EndSelect
[31]
[32]     :EndWith
       ∇
```

The CoolBar and CoolBand Objects

A CoolBar contains one or more bands (CoolBands). Each band can have any combination of a gripper bar, a bitmap, a text label, and a single child object.

Using the gripper bars, the user may drag bands from one row to another, resize bands in the same row, and maximise or minimise bands in a row. The CoolBar therefore gives the user a degree of control over the layout of the controls that it contains.

A CoolBand may not contain more than one child object, but that child object may itself be a container such as a ToolControl or a SubForm.

The following example illustrates a CoolBar containing two CoolBands, each of which itself contains a ToolControl.

```
'F'⎕WC'Form' 'CoolBar Object with ToolControls'
'F.IL'⎕WC'ImageList'('Masked' 0)('MapCols' 1)
'F.IL.'⎕WC'Bitmap'('ComCtl32' 120)⍝ STD_SMALL

'F.CB'⎕WC'CoolBar'

:With 'F.CB.C1'⎕WC'CoolBand'
    'TB'⎕WC'ToolControl'('ImageListObj' '#.F.IL')
    'TB.B1'⎕WC'ToolButton' 'New'('ImageIndex' 7)
    'TB.B2'⎕WC'ToolButton' 'Open'('ImageIndex' 8)
    'TB.B3'⎕WC'ToolButton' 'Save'('ImageIndex' 9)
:EndWith

:With 'F.CB.C2'⎕WC'CoolBand'
    'TB'⎕WC'ToolControl'('ImageListObj' '#.F.IL')
    'TB.B1'⎕WC'ToolButton' 'Cut'('ImageIndex' 1)
    'TB.B2'⎕WC'ToolButton' 'Copy'('ImageIndex' 2)
    'TB.B3'⎕WC'ToolButton' 'Paste'('ImageIndex' 3)
    'TB.B4'⎕WC'ToolButton' 'Undo'('ImageIndex' 4)
    'TB.B5'⎕WC'ToolButton' 'Redo'('ImageIndex' 5)
:EndWith
```
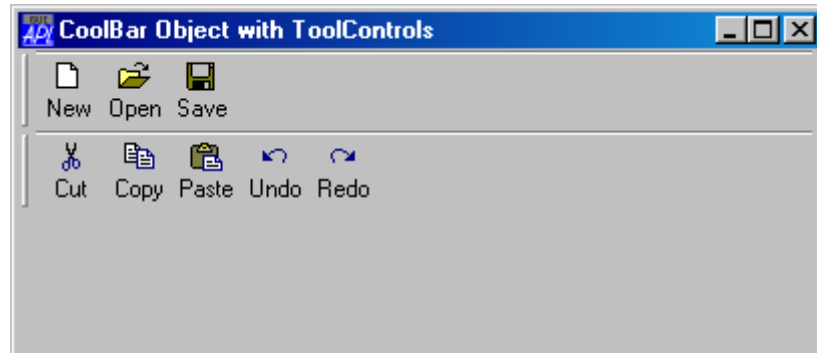
The user may move band 2 into row 1 by dragging the gripper bar:
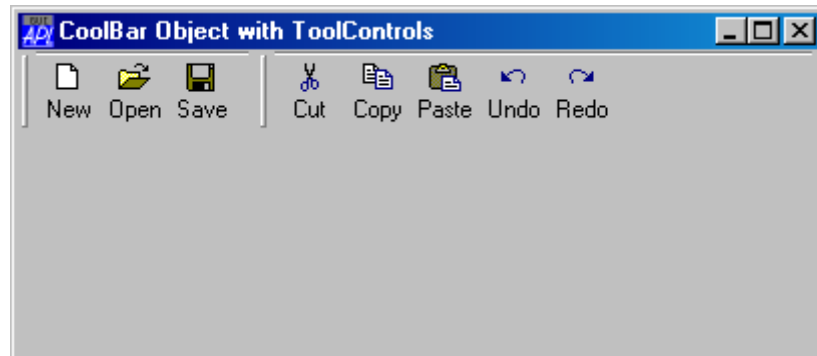
CoolBar: FixedOrder Property

FixedOrder is a property of the CoolBar and specifies whether or not the CoolBar displays CoolBands in the same order. If FixedOrder is 1, the user may move bands which have gripper bars to different rows, but the band order is static. The default is 0.

CoolBand: GripperMode Property

GripperMode is a property of a CoolBand and specifies whether or not the CoolBand has a gripper bar which is used to reposition and resize the CoolBand within its parent CoolBar. GripperMode is a character vector with the value `'Always'` (the default), `'Never'` or `'Auto'`. If GripperMode is `'Always'`, the CoolBand displays a gripper bar even if it is the only CoolBand in the CoolBar. If GripperMode is `'Never'`, the CoolBand does not have a gripper bar and may not be directly repositioned or resized by the user. If GripperMode is `'Auto'`, the CoolBand displays a gripper bar only if there are other CoolBands in the same CoolBar.

CoolBar: DblClickToggle Property

If it has a gripper bar, the user may maximise one of the bands in a row, causing the other bands to be minimised. The action required to do this is defined by the DblClickToggle property which is a property of the CoolBar.

If DblClickToggle is 0 (the default), the user must single-click the gripper bar. If DblClickToggle is 1, the user must double-click the gripper bar. These actions toggle a child CoolBand between its maximised and minimised state. The following picture shows the first CoolBand maximised.
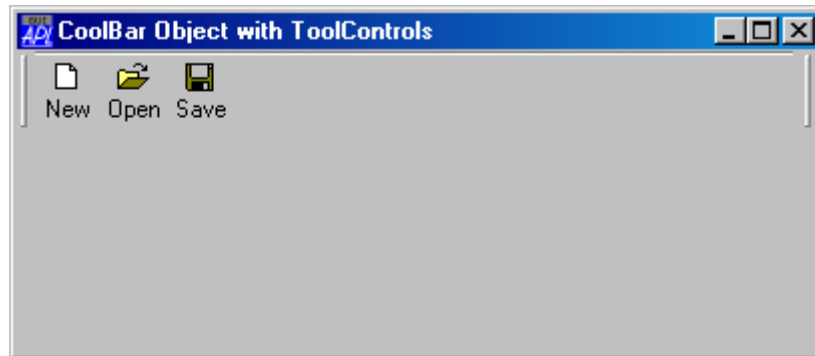
The next picture shows the second CoolBand maximised.

CoolBar: VariableHeight/BandBorders Properties

These two properties affect the appearance of the CoolBar.

The VariableHeight property specifies whether or not the CoolBar displays bands in different rows at the minimum required height (the default), or all the same height.

The BandBorders property specifies whether or not narrow lines are drawn to separate adjacent bands. The default is 0 (no lines).

The following example uses simple controls (as opposed to container controls) as children of the CoolBands and illustrate the effect of these properties on the appearance of the CoolBar.

```
'F'⎕WC'Form' 'CoolBar Object with simple controls'
'F.CB'⎕WC'CoolBar'

:With F.CB.C1'⎕WC'CoolBand'
    'E1'⎕WC'Edit' 'Edit1'
:EndWith

:With 'F.CB.C2'⎕WC'CoolBand'
    'C1'⎕WC'Combo'('One' 'Two' 'Three')('SelItems' 0 1 0)
:EndWith

:With 'F.CB.C3'⎕WC'CoolBand'
    'E2'⎕WC'Edit'(3 5⍴'Edit2')('Style' 'Multi')
:EndWith
```
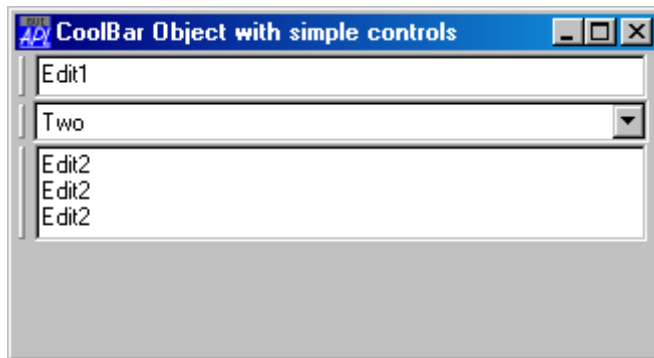
If the CoolBands are arranged in the same row, the height of the row expands to accommodate the largest one as shown below.
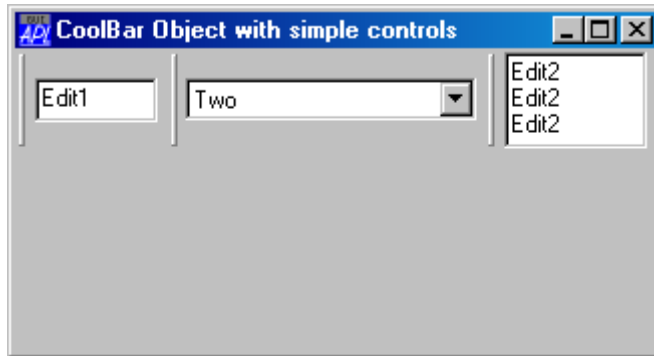
The picture below illustrates the effect of setting VariableHeight to 0.
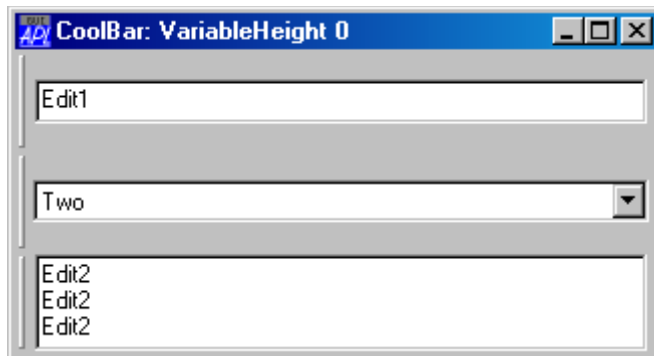
The picture below shows the affect on appearance of setting BandBorders to 1.
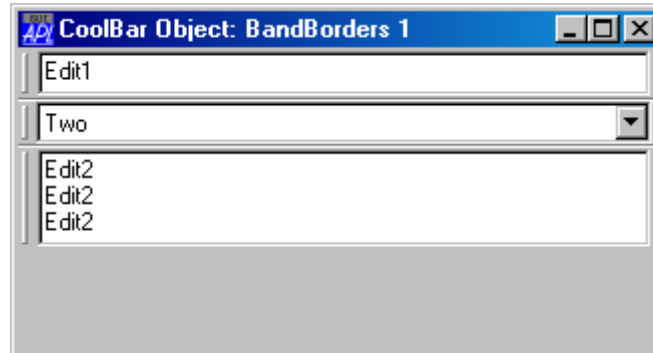
CoolBand: ChildEdge Property

ChildEdge is a property of a CoolBand and specifies whether or not the CoolBand leaves space above and below the object that it contains.

If the ChildEdge property of each CoolBand had been set to 1 in the above example, then the result would show wider borders between each band.
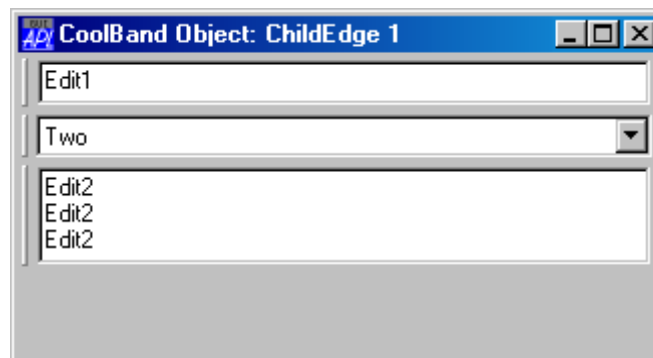
CoolBand: Caption and ImageIndex Properties

The Caption and ImageIndex properties of a CoolBand are used to display an optional text string and picture in the CoolBand.

The picture is defined by an image in an ImageList object whose name is referenced by the ImageListObj property of the parent CoolBar. The following example illustrates how this is done.

```
'F'⎕WC'Form' 'CoolBand Caption and ImageIndex'
'F.IL'⎕WC'ImageList'('Masked' 0)('MapCols' 1)
'F.IL.'⎕WC'Bitmap'('ComCtl32' 120)⍝ STD_SMALL

'F.CB'⎕WC'CoolBar'('ImageListObj' 'F.CB.IL')
'F.CB.IL'⎕WC'ImageList'('Masked' 1)('MapCols' 1)
'F.CB.IL.'⎕WC'Icon'('' 'aplicon')
'F.CB.IL.'⎕WC'Icon'('' 'editicon')

:With 'F.CB.C1'⎕WC'CoolBand' 'File'('ImageIndex' 1)
 'TB'⎕WC'ToolControl'('ImageListObj' '#.F.IL')('Divider' 0)
 'TB.B1'⎕WC'ToolButton' 'New'('ImageIndex' 7)
 'TB.B2'⎕WC'ToolButton' 'Open'('ImageIndex' 8)
 'TB.B3'⎕WC'ToolButton' 'Save'('ImageIndex' 9)
:EndWith

:With 'F.CB.C2'⎕WC'CoolBand' 'Edit'('ImageIndex' 2)
 'TB'⎕WC'ToolControl'('ImageListObj' '#.F.IL')('Divider' 0)
 'TB.B1'⎕WC'ToolButton' 'Cut'('ImageIndex' 1)
 'TB.B2'⎕WC'ToolButton' 'Copy'('ImageIndex' 2)
 'TB.B3'⎕WC'ToolButton' 'Paste'('ImageIndex' 3)
 'TB.B4'⎕WC'ToolButton' 'Undo'('ImageIndex' 4)
 'TB.B5'⎕WC'ToolButton' 'Redo'('ImageIndex' 5)
:EndWith
```
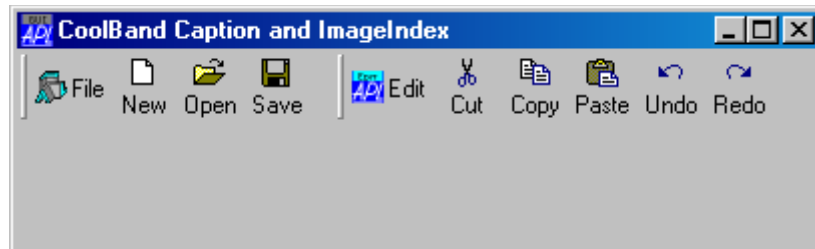
Not that the Caption and image are displayed when the CoolBand is minimised as shown below:
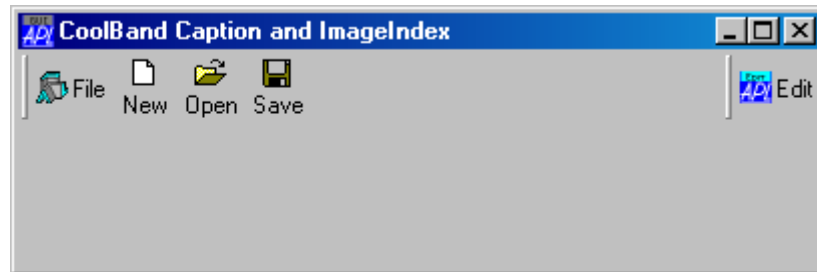
CoolBand: Size, Posn, NewLine, Index Properties

The Size property of a CoolBand is partially read-only and may only be used to specify its width; because the height of a CoolBand is determined by its contents. Furthermore, the Size property may only be specified when the CoolBand is created using `⎕WC`.

The position of a Cool Band within a CoolBar is determined by its Index and NewLine properties, and by the position and size of preceding CoolBand objects in the same CoolBar. The Posn property is read-only.

The Index property specifies the position of a CoolBand within its parent CoolBar, relative to other CoolBands and is `⎕IO` dependant. Initially, the value of Index is determined by the order in which the CoolBands are created. You may re-order the CoolBands within a CoolBar by changing its Index property with `⎕WS`.

The NewLine property specifies whether or not the CoolBand occupies the same row as an existing CoolBand, or is displayed on a new line within its CoolBar parent.

The value of NewLine in the first CoolBand in a CoolBar is always `⎕IO`, even if you specify it to be 0. You may move a CoolBand to the previous or next row by changing its NewLine property (using `⎕WS`)from 1 to 0, or from 0 to 1 respectively.

If you wish to remember the user's chosen layout when your application terminates, you must store the values of Index, Size and NewLine for each of the CoolBands. When your application is next started, you must re-create the CoolBands with the same values of these properties.

CoolBands with SubForms

The CoolBand object itself may contain only a single child object. However, if that child is a SubForm containing other objects, the CoolBand can appear to manage a group of objects. A similar effect can be obtained using a ToolBar or ToolControl.

The following example illustrates this technique. Note that the SubForms are disguised by setting their EdgeStyle and BCol properties. In addition, their AutoConf properties are set to 0 to prevent resizing of the child controls when the CoolBands are resized.

```
'F'⎕WC'Form' 'CoolBar with SubForms'('Size' 25 50)
'F'⎕WS'Coord' 'Pixel'

'F.CB'⎕WC'CoolBar'

:With 'F.CB.C1'⎕WC'CoolBand'
    'S'⎕WC'SubForm'('Size' 30 0)('EdgeStyle' 'Default')
                   ('BCol' ¯16)('AutoConf' 0)
    'S.E1'⎕WC'Edit' 'Edit 1'(2 2)(0 60)
    'S.C1'⎕WC'Combo'('One' 'Two')''(2 70)(0 60)
:EndWith

:With 'F.CB.C2'⎕WC'CoolBand'
    'S'⎕WC'SubForm'('Size' 30 0)('EdgeStyle' 'Default')
                   ('BCol' ¯16)('AutoConf' 0)
    'S.E1'⎕WC'Edit' 'Edit 2'(2 2)(0 60)
    'S.C1'⎕WC'Combo'('One' 'Two')''(2 70)(0 60)
:EndWith
```
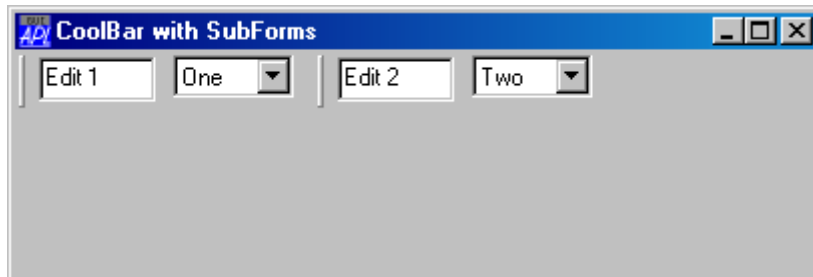
The TabControl and TabButton Objects

The TabControl object provides access to the standard Windows NT tab control.

The standard tab control is analogous to a set of dividers in a notebook and allows you to define a set of *pages* that occupy the same area of a window or dialog box. Each page consists of a set of information or a group of controls that the application displays when the user selects the corresponding tab.

A special type of tab control displays tabs that look like buttons. For example, the Windows 98 taskbar is such a tab control.

To implement a multiple page tabbed dialog, illustrated below, you should create a Form, then a TabControl with Style `'Tabs'` (which is the default) as a child of the Form.

```
'F'□WC'Form' 'TabControl: Default'('Size' 25 50)
'F.TC'□WC'TabControl'
```

Individual tabs or buttons are represented by TabButton objects which should be created as children of the TabControl object. Optional captions and pictures are specified by the Caption and ImageIndex properties of the individual TabButton objects themselves.

```
'F.TC.IL'□WC'ImageList'
'F.TC.IL.'□WC'Icon'('' 'APLIcon')
'F.TC.IL.'□WC'Icon'('' 'FUNIcon')
'F.TC.IL.'□WC'Icon'('' 'EDITIcon')
'F.TC'□WS'ImageListObj' 'F.TC.IL'
```

Next, create one or more pairs of TabButton and SubForm objects as children of the TabControl. You associate each SubForm with a particular tab by setting its TabObj property to the name of the associated TabButton object. Making the SubForms children of the TabControl ensures that, by default, they will automatically be resized correctly. (You may alternatively create your SubForms as children of the main Form and establish appropriate resize behaviour using their Attach property.)

```
'F.TC.T1'□WC'TabButton' 'One'('ImageIndex' 1)
'F.TC.T2'□WC'TabButton' 'Two'('ImageIndex' 2)
'F.TC.T3'□WC'TabButton' 'Three'('ImageIndex' 3)
'F.TC.S1'□WC'SubForm'('TabObj' 'F.TC.T1')
'F.TC.S2'□WC'SubForm'('TabObj' 'F.TC.T2')
'F.TC.S3'□WC'SubForm'('TabObj' 'F.TC.T3')
```
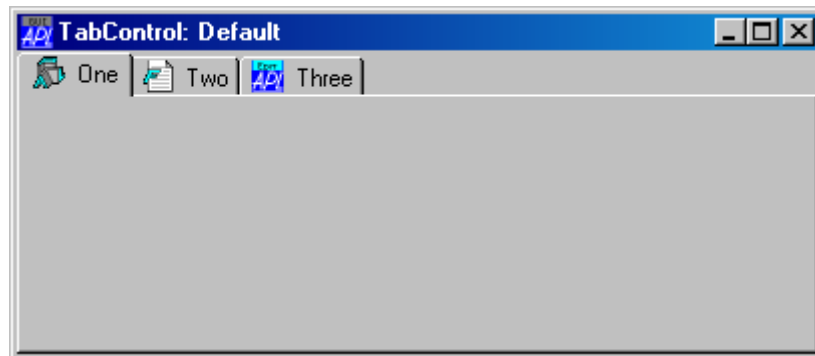
Style, FlatSeparators and HotTrack Properties

The Style property determines the overall appearance of the tabs or buttons in a TabControl and may be `'Tabs'` (the default), `'Buttons'` or `'FlatButtons'`.

A TabControl object with Style `'Buttons'` or `'FlatButtons'` may be used in a similar way (i.e. to display a set of alternative pages), although buttons in this type of TabControl are more normally used to execute commands. For this reason, these styles of TabControl are borderless.
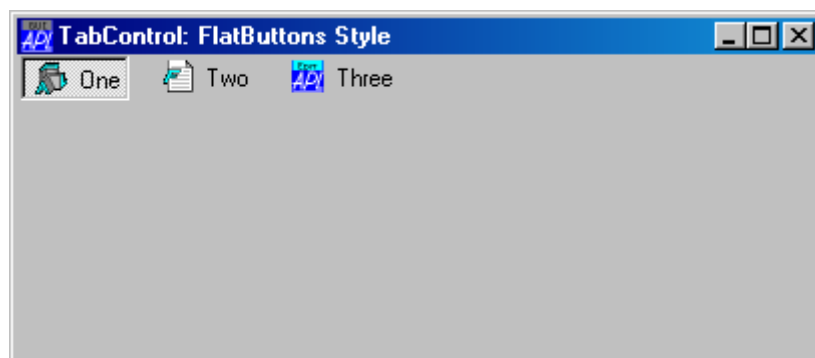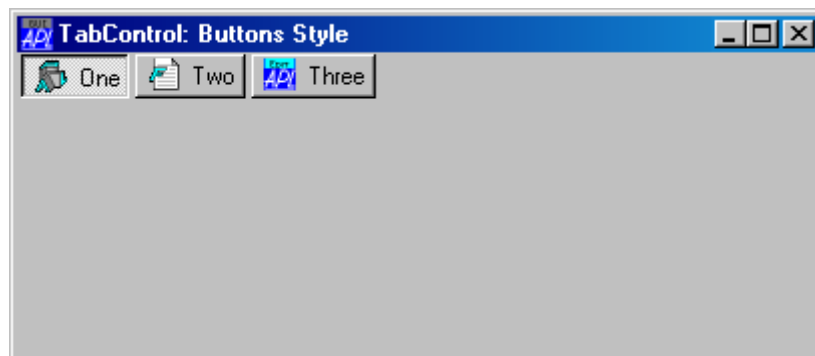
If Style is `'FlatButtons'`, the FlatSeparators property specifies whether or not separators are drawn between the buttons. The following example illustrates the effect of setting FlatSeparators to 1.
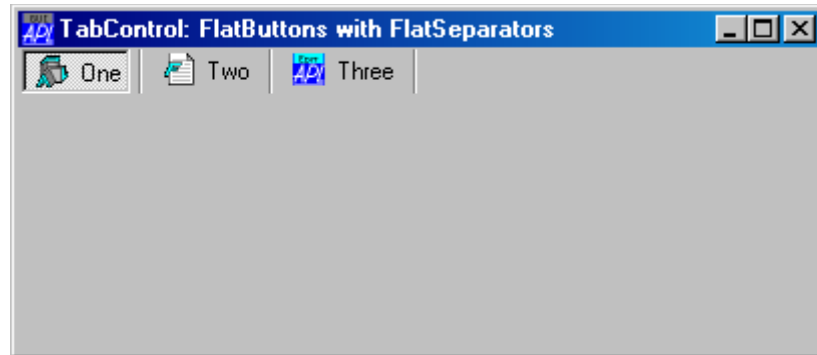
The HotTrack property specifies whether or not the tabs or buttons in a TabControl object (which are represented by TabButton objects), are automatically highlighted by the mouse pointer.

The Align Property

The Align property specifies along which of the 4 edges of the TabControl the tabs or buttons are arranged. Align also controls the relative positioning of the picture and Caption within each TabButton. Align may be Top (the default), Bottom, Left or Right.

If Align is `'Top'` or `'Bottom'`, the tabs or buttons are arranged along the top or bottom edge of the TabControl and the picture is drawn to the left of the Caption.

```
'F'⎕WC'Form' 'TabControl: Align Bottom'('Size' 25 50)
'F.TC'⎕WC'TabControl'('Align' 'Bottom')

'F.TC.IL'⎕WC'ImageList'
'F.TC.IL.'⎕WC'Icon'('' 'APLIcon')
'F.TC.IL.'⎕WC'Icon'('' 'FUNIcon')
'F.TC.IL.'⎕WC'Icon'('' 'EDITIcon')

'F.TC'⎕WS'ImageListObj' 'F.TC.IL'

'F.TC.T1'⎕WC'TabButton' 'One'('ImageIndex' 1)
'F.TC.T2'⎕WC'TabButton' 'Two'('ImageIndex' 2)
'F.TC.T3'⎕WC'TabButton' 'Three'('ImageIndex' 3)

'F.S1'⎕WC'SubForm'('TabObj' 'F.TC.T1')
'F.S2'⎕WC'SubForm'('TabObj' 'F.TC.T2')
'F.S3'⎕WC'SubForm'('TabObj' 'F.TC.T3')
```
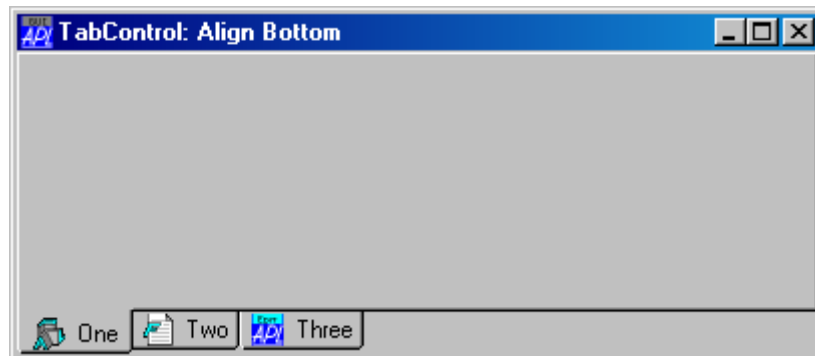
If Align is `'Left'` or `'Right'`, the tabs or buttons are arranged top-to-bottom along the left or right edge of the TabControl as shown below.
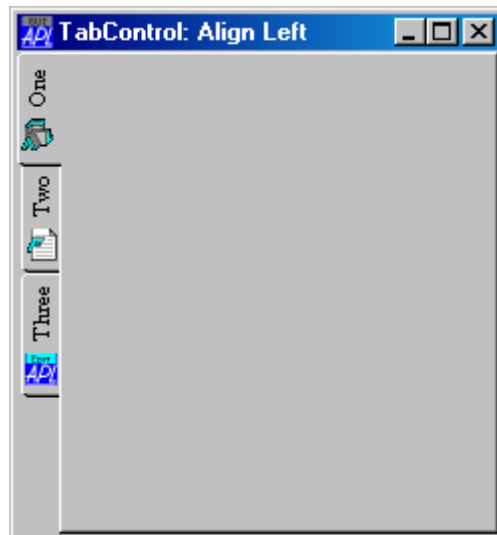
The MultiLine Property

The MultiLine property of a TabControl determines whether or not your tabs or buttons will be arranged in multiple flights or multiple rows/columns.

The default value of MultiLine is 0, in which case, if you have more tabs or buttons than will fit in the space provided, the TabControl displays an UpDown control to permit the user to scroll them.
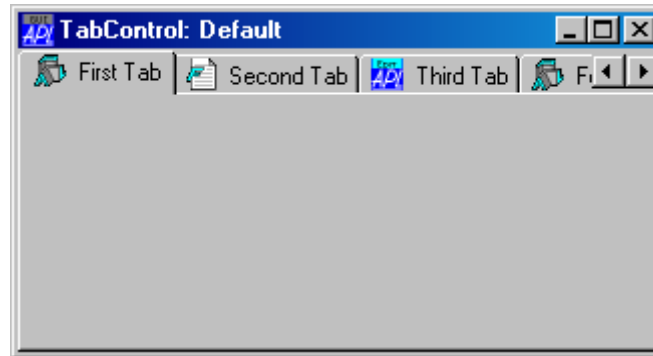
If MultiLine is set to 1, the tabs are displayed in multiple flights.

If the TabControl has Style `'Buttons'` and MultiLine is set to 1, the buttons are displayed in multiple rows.
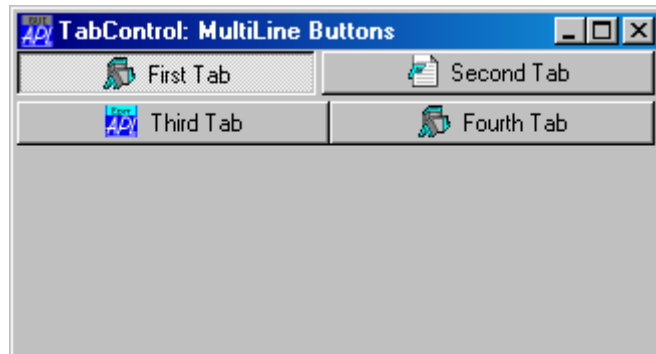
The ScrollOpposite Property

The ScrollOpposite property specifies that unneeded tabs scroll to the opposite side of a TabControl, when a tab is selected. Setting ScrollOpposite to 1 forces MultiLine to 1 too.

The following example illustrates a TabControl with ScrollOpposite set to 1, after the user has clicked *Third Tab*. Notice that, in this example, the SubForms have been created as children of the TabControl. This is necessary to ensure that they are managed correctly in this case.

```
'F'⎕WC'Form' 'TabControl: ScrollOpposite'('Size' 25 40)
'F.TC'⎕WC'TabControl'('ScrollOpposite' 1)

'F.TC.IL'⎕WC'ImageList'
'F.TC.IL.'⎕WC'Icon'('' 'APLIcon')
'F.TC.IL.'⎕WC'Icon'('' 'FUNIcon')
'F.TC.IL.'⎕WC'Icon'('' 'EDITIcon')

'F.TC'⎕WS'ImageListObj' 'F.TC.IL'

'F.TC.T1'⎕WC'TabButton' 'First Tab'('ImageIndex' 1)
'F.TC.T2'⎕WC'TabButton' 'Second Tab'('ImageIndex' 2)
'F.TC.T3'⎕WC'TabButton' 'Third Tab'('ImageIndex' 3)
'F.TC.T4'⎕WC'TabButton' 'Fourth Tab'('ImageIndex' 1)
```
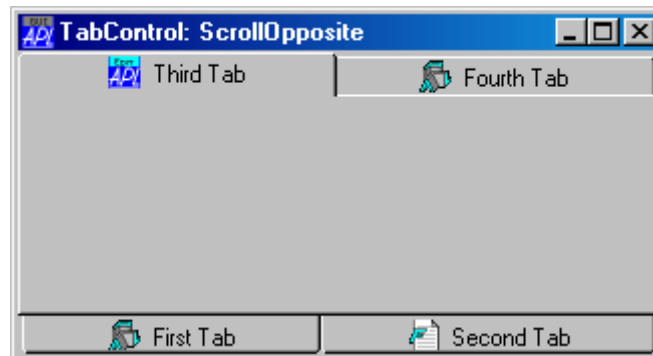
If MultiLine is 1, the way that multiple flights of tabs or rows/columns of buttons are displayed is further defined by the Justify property which may be `'Right'` (the default) or `'None'`.

The Justify Property

If Justify is `'Right'` (which is the default), the TabControl increases the width of each tab, if necessary, so that each row of tabs fills the entire width of the tab control. Otherwise, if Justify is empty or `'None'`, the rows are ragged as shown below.

```
'F'⎕WC'Form' 'TabControl: MultiLine Tabs, Justify None'
'F.TC'⎕WC'TabControl'('MultiLine' 1)('Justify ' 'None')
'F.TC.IL'⎕WC'ImageList'
'F.TC.IL.'⎕WC'Icon'('' 'APLIcon')
'F.TC.IL.'⎕WC'Icon'('' 'FUNIcon')
'F.TC.IL.'⎕WC'Icon'('' 'EDITIcon')
'F.TC'⎕WS'ImageListObj' 'F.TC.IL'
'F.TC.T1'⎕WC'TabButton' 'First Tab'('ImageIndex' 1)
'F.TC.T2'⎕WC'TabButton' 'Second Tab'('ImageIndex' 2)
'F.TC.T3'⎕WC'TabButton' 'Third Tab'('ImageIndex' 3)
'F.TC.T4'⎕WC'TabButton' 'Fourth Tab'('ImageIndex' 1)
```
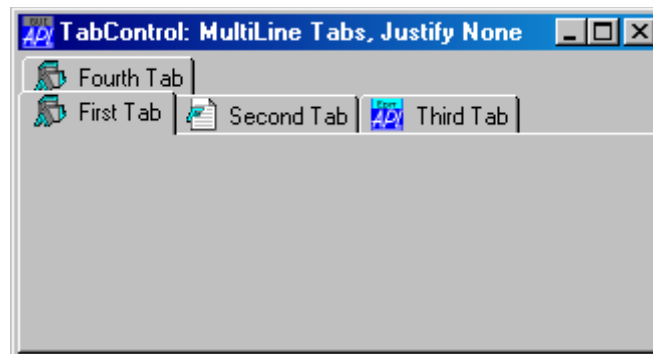
The next picture illustrates the effect of Justify `'None'` on a TabControl with Style `'Buttons'`.
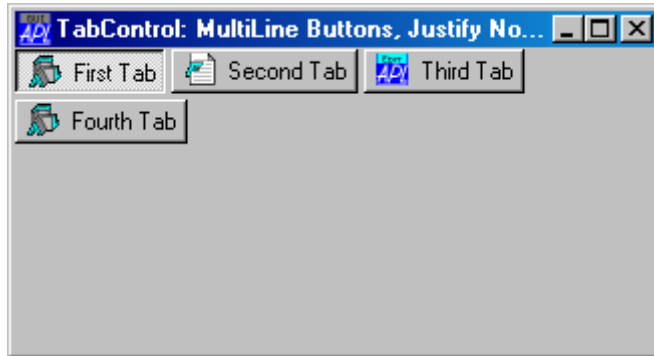
The TabSize and TabJustify Properties

By default, the size of the tabs may vary from one row to another. Fixed size tabs may be obtained by setting the TabSize property.

If fixed size tabs are in effect, the positions at which the picture and Caption are drawn within each TabButton is controlled by the TabJustify property which may be `'Centre'`(the default), `'Edge'`, or `'IconEdge'`.

```
'F'⎕WC'Form' 'TabControl: TabJustify Centre'('Size' 10 40)
'F.TC'⎕WC'TabControl'('Style' 'Buttons')('TabSize'0 30)

'F.TC.IL'⎕WC'ImageList'
'F.TC.IL.'⎕WC'Icon'('' 'APLIcon')
'F.TC.IL.'⎕WC'Icon'('' 'FUNIcon')
'F.TC.IL.'⎕WC'Icon'('' 'EDITIcon')
'F.TC'⎕WS'ImageListObj' 'F.TC.IL'

'F.TC.T1'⎕WC'TabButton' 'One'('ImageIndex' 1)
'F.TC.T2'⎕WC'TabButton' 'Two'('ImageIndex' 2)
'F.TC.T3'⎕WC'TabButton' 'Three'('ImageIndex' 3)
```
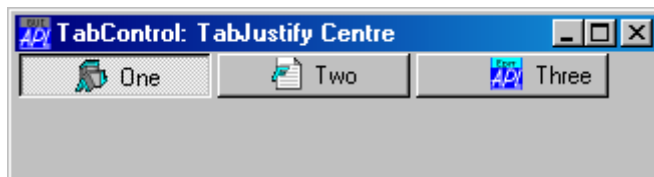
If TabJustify is set to `'Edge'` then the picture and text on the TabButton are justified along the side defined by the Align property (default `'Top'`).

```
'F'⎕WC'Form' 'TabControl: TabJustify Edge'('Size' 10 40)
'F.TC'⎕WC'TabControl'('Style' 'Buttons')
                     ('TabJustify' 'Edge')('TabSize'⍬ 30)
```

etc.
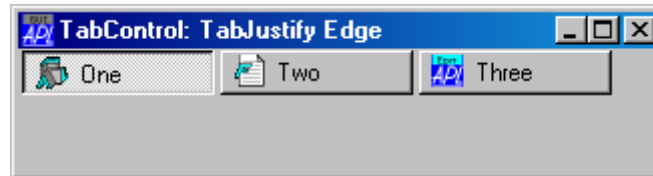
If, instead, the TabJustify property is set to `'IconEdge'` then the text is centred and only the icons are justified.
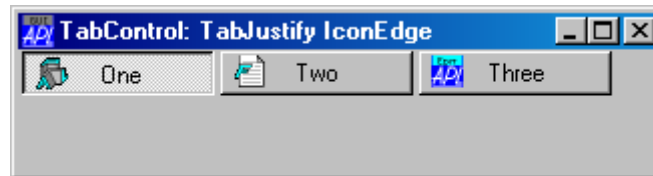
The TabFocus Property

The TabFocus property specifies the focus behaviour for the TabControl object.

TabFocus is a character vector that may be `'Normal'` (the default), `'Never'` or `'ButtonDown'`.

If TabFocus is `'Normal'`, the tabs or buttons in a TabControl do not immediately receive the input focus when clicked, but only when clicked a second time. This means that, normally, when the user circulates through the tabs, the input focus will be given to the appropriate control in the associated SubForm. However, if the user clicks twice in succession on the same tab or button, the TabControl itself will receive the input focus.

If TabFocus is `'ButtonDown'`, the tabs or buttons in a TabControl receive the input focus when clicked.

If TabFocus is `'Never'`, the tabs or buttons in a TabControl *never* receive the input focus. This allows the user to circulate through a set of tabbed SubForms without ever losing the input focus to the TabControl itself.

The StatusBar Object

Like the Toolbar, the StatusBar object is also a container that manages its children. However, the StatusBar may contain only one type of object, namely StatusFields. By default, the StatusBar is a flat grey object, positioned along the bottom edge of a Form, upon which the StatusFields are drawn as sunken rectangles. StatusFields display textual information and are typically used for help messages and for monitoring the status of an application. They can also be used to automatically report the status of the Caps Lock, Num Lock, Scroll Lock, and Insert keys

The following example illustrates a default StatusBar containing three StatusFields. Notice how the StatusFields are positioned automatically.

```
'TEST'⎕WC'Form' 'Simple StatusBar' ('EdgeStyle' 'Default')
'TEST.SB'⎕WC'StatusBar'
'TEST.SB.S1'⎕WC'StatusField' 'Field1:' 'text1'
'TEST.SB.S2'⎕WC'StatusField' 'Field2:' 'text2'
'TEST.SB.S3'⎕WC'StatusField' 'Field3:' 'text3'
```
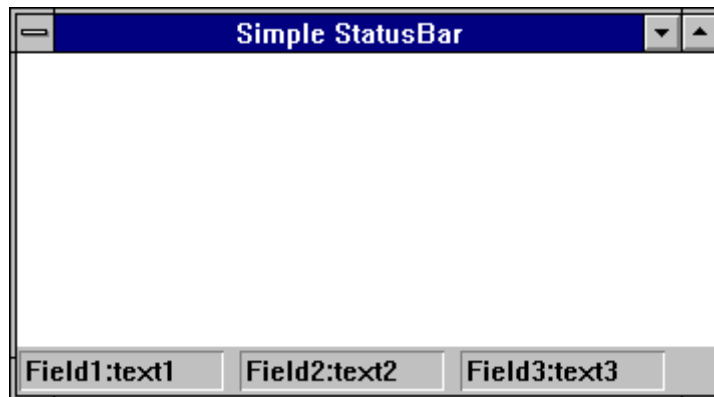

Figure 4.11   A Default StatusBar

The following example illustrates a scrolling StatusBar. The fourth StatusField extends beyond the right edge of the StatusBar and, because HScroll is ‾2, a mini scrollbar appears. The result is shown in Figure 6.11.

```
'TEST'⎕WC'Form' 'Scrolling StatusBar'
              ('EdgeStyle' 'Default')
              ('Coord' 'Pixel') ('Size' 150 330)

'TEST.SB'⎕WC'StatusBar'('HScroll' ‾2)

'TEST.SB.S1'⎕WC'StatusField' 'Field1:' 'text1'
'TEST.SB.S2'⎕WC'StatusField' 'Field2:' 'text2'
'TEST.SB.S3'⎕WC'StatusField' 'Field3:' 'text3'
'TEST.SB.S4'⎕WC'StatusField' 'Field4:' 'text4'
```
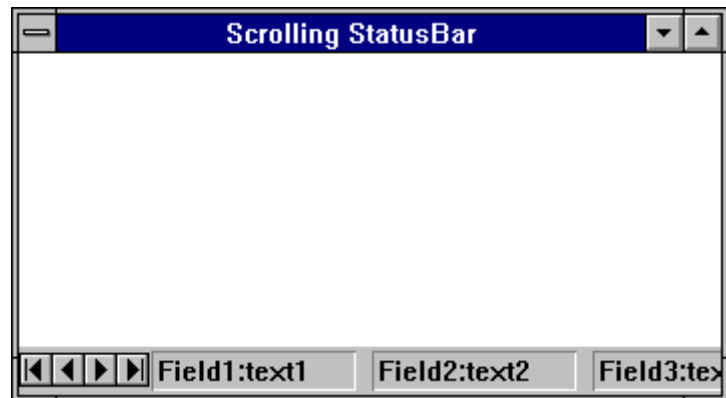

Figure 4.12   A Scrolling StatusBar

As an alternative to single-row scrolling StatusBar, you can have a multi-line one. Indeed, this is the default if you omit to specify HScroll. However, you do have to explicitly set the height of the StatusBar to accommodate the second row.

```
'TEST'⎕WC'Form' 'Multi Line StatusBar'
              ('EdgeStyle' 'Default')
              ('Coord' 'Pixel') ('Size' 150 330)

'TEST.SB'⎕WC'StatusBar'('Size' 48 0)

'TEST.SB.S1'⎕WC'StatusField' 'Field1:' 'text1'
'TEST.SB.S2'⎕WC'StatusField' 'Field2:' 'text2'
'TEST.SB.S3'⎕WC'StatusField' 'Field3:' 'text3'
'TEST.SB.S4'⎕WC'StatusField' 'Field4:' 'text4'
```
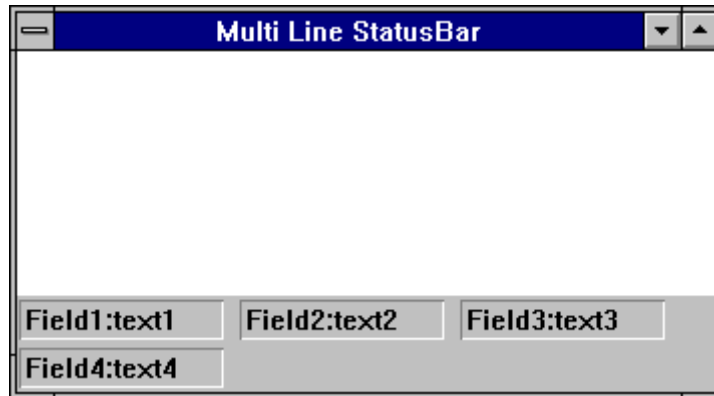

Figure 4.13   A Multi-line StatusBar

Using StatusFields

There are basically three ways of using StatusFields. Firstly, you can display information in them directly from your program by setting their Caption and/or Text properties. For example, if you are executing a lengthy calculation, you may wish to display the word "Calculating ..." as the Caption of a StatusField and, as the calculations proceed, display (say) "Phase 1" followed in due course by "Phase 2", and so forth. You can also use StatusFields to display application messages, including warning and error messages, where the use of a MsgBox is inappropriate.

The second major use of a StatusField is to display **hints** which you do by setting the HintObj property of an object to the name of the StatusField. Used in this way, a StatusField automatically displays context sensitive help when the user places the mouse pointer over an object. This topic is described in Chapter 5.

The third use of a Status Field is to monitor the status of the keyboard. This is achieved by setting its Style property to one of the following keywords:

Keyword	Meaning
CapsLock	Monitors state of Caps Lock key
ScrollLock	Monitors state of Scroll Lock key
NumLock	Monitors state of Num Lock key
KeyMode	Monitors the keyboard mode (APL/ASCII (Classic Edition only)
InsRep	Monitors the state of the Insert/Replace toggle key

The following example illustrates different uses of the StatusField object. The first StatusField `F.SB.S1` is used for context-sensitive help by making it the HintObj for the Form `F`. The second StatusField `F.SB.S2` is simply used to display application status; in this case "Ready ...". The third and fourth StatusField objects monitor the status of the Insert and Caps Lock keys respectively. Note that whilst the Caps Lock, Num Lock and Scroll Lock keys have a recognised *state*, the Insert key does not. Initially, APL sets the key to "Ins" and then toggles to and from "Rep" whenever the key is pressed. To discover which mode the keyboard is in, you should use ⎕WG to read the value of the Text property of the StatusField.

```
'F'⎕WC'Form' 'Using StatusFields'(60 40)(25 50)

'F.SB'⎕WC'StatusBar'('Coord' 'Pixel')

'F.SB.S1'⎕WC'StatusField'('Size'⍬ 150)
'F'⎕WS'HintObj' 'F.SB.S1'

'F.SB.S2'⎕WC'StatusField' 'Ready ...'
'F.SB.S3'⎕WC'StatusField'('Style' 'InsRep')('Size'⍬ 50)
'F.SB.S4'⎕WC'StatusField'('Style' 'CapsLock')('Size'⍬ 50)

'F.L'⎕WC'List'WINES(0 0)(80 100)('Hint' 'Choose a Wine')
```
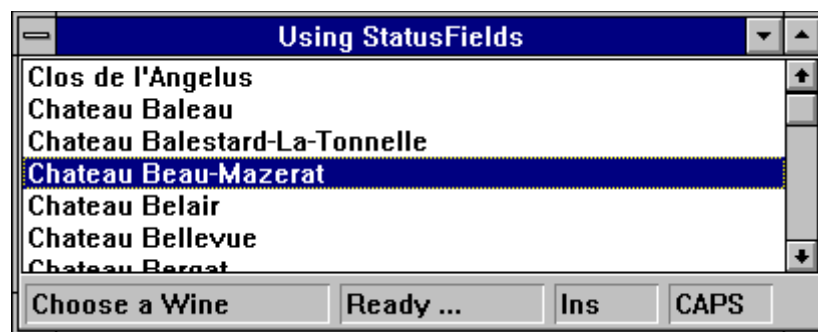
CHAPTER 5

Hints and Tips

In many applications it is often a good idea to provide short context-sensitive help messages that tell the user what action each control (menuitem, button and so forth) performs. It is conventional to do this by displaying a message when the user points to a control with the mouse. The provision of this facility is particularly helpful for users who are not familiar with your application or who use it only occasionally. Constant prompting can however become irritating for an experienced user, so it is a good idea to provide a means to disable it.

Dyalog APL/W provides two mechanisms, *hints* and *tips*, that make the provision of context-sensitive help very easy and efficient to implement. **Hints** are help messages displayed in a fixed region, typically a field in a status bar, that is reserved for the purpose. For example, when the user browses through a menu, a message describing each of the options may be displayed in the status bar. The user has only to glance at the status bar to obtain guidance. **Tips** are similar, but instead of being displayed in a fixed location, they are displayed as pop-up messages over the control to which they refer. The choice of using hints or tips is a matter of taste and indeed many applications feature both.

Using Hints

All of the GUI objects supported by Dyalog APL that have a visible presence on the screen have a Hint property and a HintObj property. Quite simply, when the user moves the mouse pointer over the object the contents of its Hint property are displayed in the object referenced by its HintObj property. When the user moves the mouse pointer away from the object, its Hint disappears. If an object has a Hint, but its HintObj property is empty, the system uses the HintObj defined for its parent, or for its parent's parent, and so forth up the tree. If there is no HintObj defined, the Hint is simply not displayed. This mechanism has two useful attributes:

- it allows you to easily define a single region for help messages for all of the controls in a Form, but still provides the flexibility for using different message locations for different controls if appropriate.

- to enable or disable the display of hints all you typically have to do is to set or clear the HintObj property on the parent Form

The object named by HintObj may be any object with either a Caption property or a Text property. Thus you can use the Caption on a Label, Form, or Button or the text in an Edit object. If you use a StatusField object which has both Caption *and* Text properties, the Text property is employed. If you set HintObj to the name of an object which possesses neither of these properties, the hints will simply not be displayed. The following example illustrates the use of a StatusField for displaying hints.

Example: Using a StatusField for Hints

This example illustrates the use of a StatusField object to display hints. The result of the code shown below is illustrated in Figure 5.1.

```
'Test'⎕WC 'Form' 'Using Hints'('HintObj' 'Test.SB.H')

'Test.MB' ⎕WC 'MenuBar'
'Test.MB.F' ⎕WC 'Menu' '&File'
HINT ← 'Creates a new empty document'
'Test.MB.F.New' ⎕WC 'MenuItem' '&New' ('Hint' HINT)

'Test.SB' ⎕WC 'StatusBar'
'Test.SB.H' ⎕WC 'StatusField' ('Size' 0 98)
```
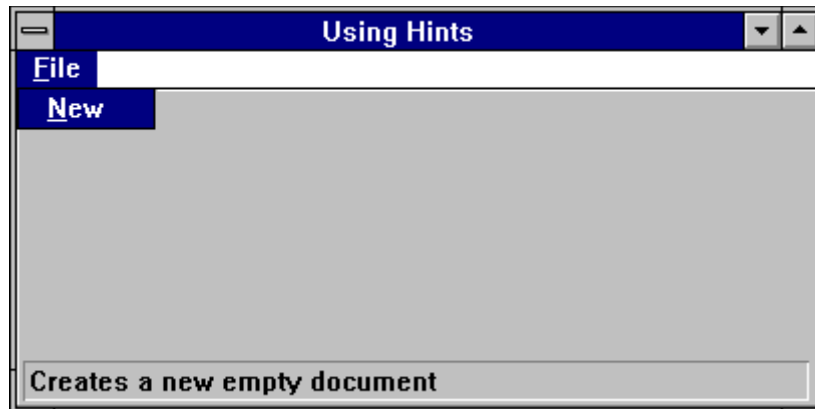

Figure 5.1   Using a StatusBar to display Hints

Example: Using an Edit Object for Hints

You can display a much larger amount of information using a multi-line Edit object as shown in this example.

```
'Test'⎕WC 'Form' 'Using Hints' ('HintObj'
                                 'Test.ED')

'Test.MB' ⎕WC 'MenuBar'
'Test.MB.F' ⎕WC 'Menu' '&File'
HINT ← 100ρ'Creates a new empty document '
'Test.MB.F.New' ⎕WC 'MenuItem' '&New' ('Hint' HINT)

'Test.ED' ⎕WC 'Edit' ('Style' 'Multi')
```

The result of this example is illustrated in Figure 5.2.
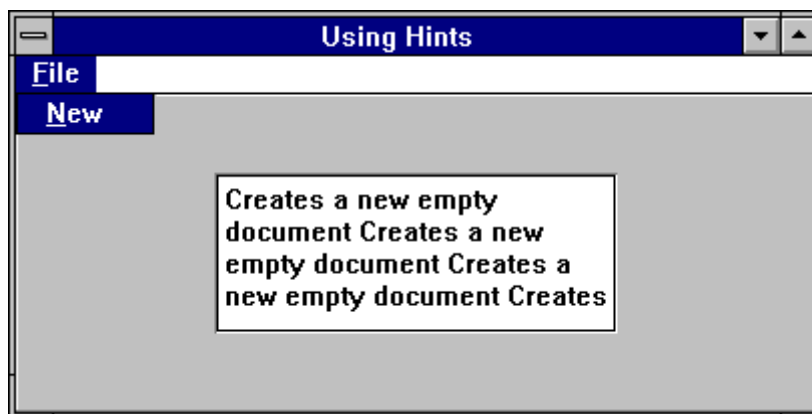
Figure 5.2   Displaying Hints in an Edit object

Using Tips

Tips work in a very similar way to Hints. Most of the GUI objects that have a visible presence on the screen have a Tip property and a TipObj property. Exceptions are Menus, MenuItems and other pop-up objects. The TipObj property contains the name of a TipField object. This is a special kind of pop-up object whose sole purpose is to display tips. When the user moves the mouse pointer over the object the corresponding TipField appears displaying the object's Tip. When the mouse pointer moves away from the object, the TipField disappears. If an object has a Tip, but its TipObj property is empty, the system uses the TipObj defined for its parent, or for its parent's parent, and so forth up the tree. If there is no TipObj defined, the Tip is simply not displayed. Normally, you need only define one TipField for your application, but if you want to use different colours or fonts for individual tips, you may define as many different TipFields as you require. Again, it is very simple to turn tips on and off.

Example

```
This example shows how easy it is to associate a tip with
an object, in this case a Button.
```

```
'Test'⎕WC 'Form' 'Using Tips'('TipObj' 'Test.Tip')
'Test.Tip' ⎕WC 'TipField'
'Test.B' ⎕WC 'Button' '&Ok' ('Tip' 'Press Me')
```

Figure 5.3   Using Tips

Hints and Tips Combined

There is no reason why you cannot provide Hints *and* Tips. Example 4 shows how an object, in this case a Combo, can have both defined.

Example

```
'Test'⎕WC 'Form' 'Using Hints and Tips'

'Test.SB' ⎕WC 'StatusBar'
'Test.SB.H' ⎕WC 'StatusField' ('Size' 0 98)
'Test' ⎕WS 'HintObj' 'Test.SB.H'

'Test.Tip' ⎕WC 'TipField'
'Test' ⎕WS 'TipObj' 'Test.Tip'

'Test.C' ⎕WC 'Combo' WINES
'Test.C' ⎕WS 'Hint' 'Select your wine from this
                     list'
'Test.C' ⎕WS 'Tip' 'Wine Cellar'
```
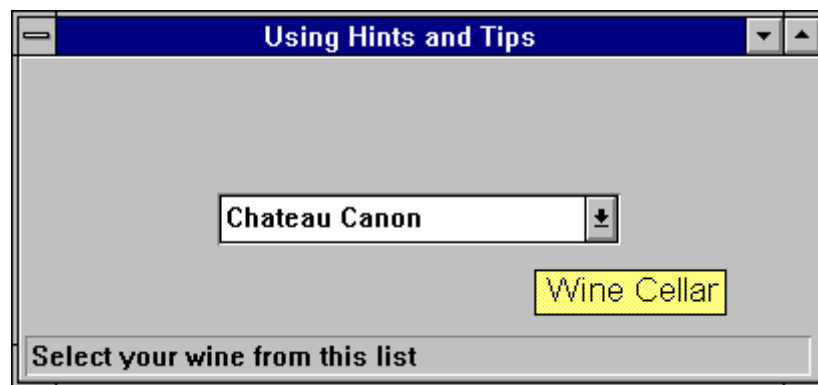

Figure 5.4   Hints and Tips Combined

C H A P T E R   6

Using the Grid Object

The Grid object allows you to display information in a series of rows and columns and lets the user input and change the data. The Grid has four main components; a matrix of cells that represents the data, a set of row titles, a set of column titles, and a pair of scroll bars. Figure 7.1 illustrates these components. The scroll bars scroll the data cells and either the row or column titles. The row titles remain fixed in place when the data cells scroll horizontally and the column titles stay fixed when the data is scrolled vertically.
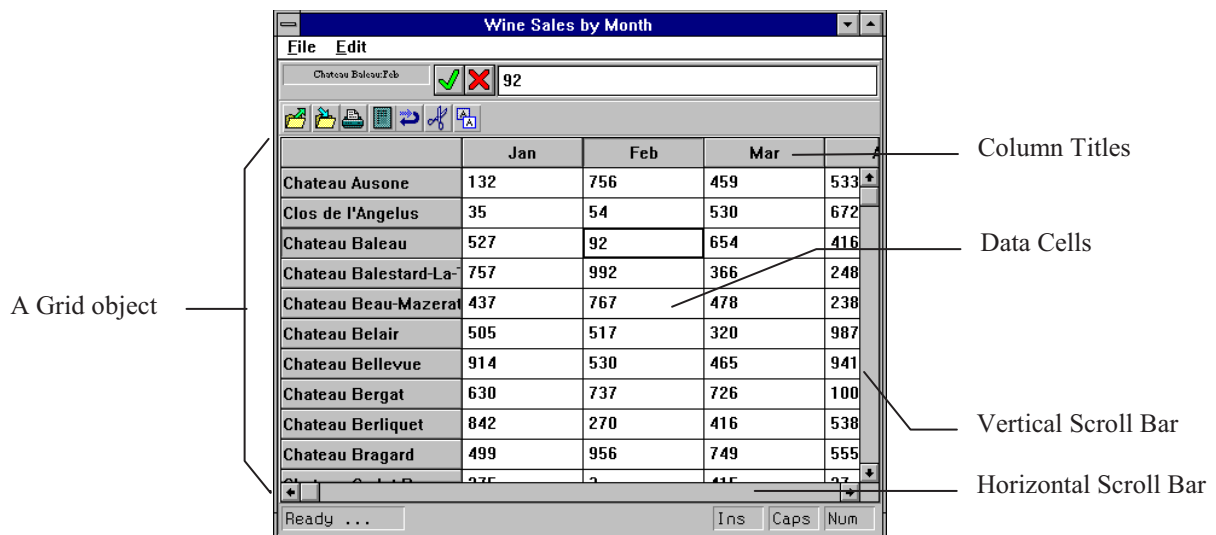
Figure 7.1   The components of the Grid object

Defining the Grid Layout

The data in the Grid is specified by its Values property. This is a matrix whose elements are displayed in the corresponding cells in the Grid. An element (and therefore a cell) may contain a single number, a single character, a character vector or a character matrix.

The row and column titles are specified by the RowTitles and ColTitles properties respectively. Each of these is a vector whose elements may be character vectors, character matrices, or vectors of character vectors. Matrices and vectors of vectors specify multi-line titles. The specification of RowTitles and ColTitles is optional. If they are omitted, the system will display standard spreadsheet defaults; 1,2,3... for the rows and A,B,C...AA,AB...for the columns. By default, RowTitles are left-justified; ColTitles are centred, although this can be changed using the RowTitleAlign and ColTitleAlign properties. Multi-level titles may be defined using the RowTitleDepth and ColTitleDepth properties.

The height of the column headers and the width of the row titles are defined by the TitleHeight and TitleWidth properties. To disable the display of column and/or row titles, you simply set TitleHeight and/or TitleWidth to 0.

The dimensions of the cells of the Grid are defined by the CellHeights and CellWidths properties. Each of these may be a scalar which applies to all rows and columns, or a vector which specifies the heights and widths of each row and column individually. You may also allow the user to resize rows and columns with the mouse.

The Grid shown in Figure 7.1 was created using the following few lines of APL code. Note that by default a Grid is positioned in the centre of a Form and occupies a quarter of its area. In this example, the Grid is positioned at (0 0) and made to occupy the entire client area of the Form. Note that SALES is a simple numeric matrix, WINES is a vector of character vectors containing the names of different wines, and MONTHS is a similar vector of month names.

```
'Test' ⎕WC 'Form' ('Coord' 'Pixel')

'Test.G' ⎕WC 'Grid' SALES (0 0) ('Test' ⎕WG 'Size')

'Test.G' ⎕WS 'RowTitles' WINES
'Test.G' ⎕WS 'ColTitles' MONTHS
'Test.G' ⎕WS 'TitleWidth' 150
```

Defining Overall Appearance

By default, the Grid inherits its font from the parent Form, or ultimately, from the Root object. This defaults to your Windows System font.

You can change the font for the Grid as a whole using its FontObj property. This font will be used for the row titles, column titles and for the data. You can separately define the font for the data using the CellFonts property. Thus, for example, if you wanted to use Helvetica 12 for the titles and Arial 10 for the data, you could do so as follows:

```
'Test.G'  ⎕WS 'FontObj' 'Helvetica' 12

'Test.CF' ⎕WC 'Font' 'Arial' 10
'Test.G'  ⎕WS 'CellFonts' 'Test.CF'
```

The FCol and BCol properties specify the foreground and background colours for the text in the data cells. The default colour scheme is black on white. FCol and BCol may define single colours which refer to all the cells, or a set of colours to be applied to different cells

The colour of the gridlines is specified by GridFCol. To draw a Grid with no gridlines, set GridFCol to the same colour as is defined by BCol.

If the Grid is larger than the space occupied by the data cells, GridBCol specifies the colour used to fill the area between the end of the last column of data and the right edge of the Grid, and between the bottom row of data and the bottom edge of the Grid.

The ClipCells property determines whether or not the Grid displays partial cells. The default is 1. If you set ClipCells to 0, the Grid displays only complete cells and automatically fills the space between the last visible cell and the edge of the Grid with the GridBCol colour.

The following example shows a default Grid (ClipCells is 1) in which the third column of data is in fact incomplete (clipped), although this is by no means apparent to the user.

ClipCells is 1 (the default)			
	A	B	C
1	13	75	4
2	51	83	
3	68	58	9
4	76	26	
5	98	72	7

This second picture shows the effect on the Grid of setting ClipCells to 0 which prevents such potential confusion.

ClipCells is 0		
	A	B
1	13	75
2	51	83
3	68	58
4	76	26
5	98	72

Row and Column Titles

Row and column titles are defined by the RowTitles and ColTitles properties, each of which is a vector of character arrays. An element of RowTitles and ColTitles may be a character vector specifying a 1-row title, or a matrix or vector of vectors which specify multi-row titles.

The height of the area used to display column titles is specified by the TitleHeight property. The width of the area used to display row titles is defined by the TitleWidth property. The alignment of text within the title cells is defined by RowTitleAlign and ColTitleAlign and the colour of the text is specified by RowTitleFCol and ColTitleFCol.

Multi-level titles are also possible and are defined by the RowTitleDepth and ColTitleDepth properties. An example of what can be achieved is shown below.

			Wine		Champagne			Beer and Cider
			Red	White	Red	White	Rose	
1995	Q1	Jan	60	92	16	4	72	52
		Feb	41	6	2	7	6	12
		Mar	98	39	25	2	37	42
	Q2	Apr	83	95	32	55	34	90
		May	65	54	37	23	16	92
		Jun	5	23	61	23	57	19
	Q3	Jul	42	19	64	79	70	4
		Aug	25	54	84	47	65	97
		Sep	73	91	34	58	97	97
	Q4	Oct	60	65	3	50	56	75
		Nov	30	39	50	45	31	93
		Dec	1	32	76	99	43	10

Displaying and Editing Values in Grid Cells

The Grid can display the value in a cell directly (as in Fig 7.1) or indirectly via an *associated object*. You do not (as you might first expect) define input and validation characteristics for the cells directly, instead you do so *indirectly* through associated objects. Objects are associated with Grid cells by the Input property. If a cell has an associated object, its value is displayed and edited using that object. Several types of object may be associated with Grid cells, including Edit, Label, Button (Push, Radio and Check), and Combo objects. You can use a single associated object for the entire Grid, or you can associate different objects with individual cells.

Edit and Label objects impose formatting on the cells with which they are associated according to the values of their FieldType and Decimal properties (for numbers, dates and time) and their Justify property (for text). In addition, Label objects protect cells (because a Label has no input mechanism), while Edit objects impose input validation. If you use an Edit object with a FieldType of Numeric, the user may only enter numbers into the corresponding cells of the Grid. For both Edit and Label objects, the FieldType and Decimals properties of the object are used to format the data displayed in the corresponding cells of the Grid. For example, if the FieldType property of the associated object is Date, the numeric elements in Values will be displayed as dates.

Numeric cells may also be formatted using the FormatString property which applies ⎕FMT format specifications to the data. The AlignChar property permits formatted data to be aligned in a column. For example, you can specify that numbers in a column are aligned on their decimal points.

Combo objects can be used to allow the user to select a cell value from a set of alternatives. Radio and Check Buttons may be used to display and edit Boolean values.

Associated Edit, Label and Combo objects may be *external* to the Grid (for example, you can have the user type values into a companion edit field) or they may be *internal*. Internal objects (which are implemented as children of the Grid) float from cell to cell and allow the data to be changed *in-situ*. Button, Spinner and TrackBar objects may only be internal.

Using a Floating Edit Field

If the Edit object specified by Input is owned by (i.e. is a child of) the Grid itself, the Edit object *floats* from cell to cell as the user moves around the Grid. For example, if the user clicks on the cell addressed by row 4, column 3, the Edit object is automatically moved to that location and the data in that cell is copied into it ready for editing. When the user moves the focus away from this cell, the data in the Edit object is copied back into it (and into the corresponding element of the Values property) before the Edit object is moved away to the new cell location. This mechanism provides *in-situ* editing. Continuing the example illustrated in Figure 7.1, in-situ editing could be achieved as follows:

```
'Test.G.ED' ⎕WC 'Edit' ('FieldType' 'Numeric')
'Test.G'    ⎕WS 'Input' 'Test.G.ED'
```

In-situ editing provides two input modes; Scroll and InCell. In Scroll mode the cursor keys move from one cell to another. In InCell mode, the cursor keys move the cursor a character at a time within the cell; to switch to a new cell, the user must press the Tab key or use the mouse. The InputMode property allows you to control the input mode directly or to allow the user to switch from one to another. In the latter case, the user does so by pressing a key defined by the InputModeKey property or by double-clicking the left mouse button.

Using a Fixed Edit Field

A different style of editing may be provided by specifying the name of an external Edit object that you have created. This can be any Edit object you wish to use; it need not even be owned by the same Form as the Grid. In this case, the Edit object remains stationary (wherever you have positioned it), but as the user moves the focus from cell to cell, the cell contents are copied into it and made available for editing. The current cell is identified by a thick border. When the user shifts the focus , the data is copied out from the Edit object into the corresponding cell before data in the newly selected one is copied in. Continuing the example illustrated in Figure 7.1, external editing could be achieved as follows:

```
'Test.ED'   ⎕WC 'Edit' ('FieldType' 'Numeric')
'Test.G'    ⎕WS 'Input' 'Test.ED'
```

Using Label Objects

If Input specifies a Label object, it too may either be a child of the Grid or an external Label. A Label is useful to format cell data (through its FieldType property) and to protect cells from being changed

If the Label is a child of the Grid, it floats from cell to cell in the same way as a floating Edit object. However, unlike the situation with other objects, the row and column titles are not indented to help identify the current cell. If the Label is borderless (which is the default) and has the same font and colour characteristics of the cells themselves, the user will receive no visual feedback when a corresponding cell is addressed, even though the current cell (reflected by the CurCell property) does in fact change. Therefore, if you want to protect the data by using a Label *and* you want the user to be able to identify the current cell, you should give the Label a border, a special colour scheme or a special font.
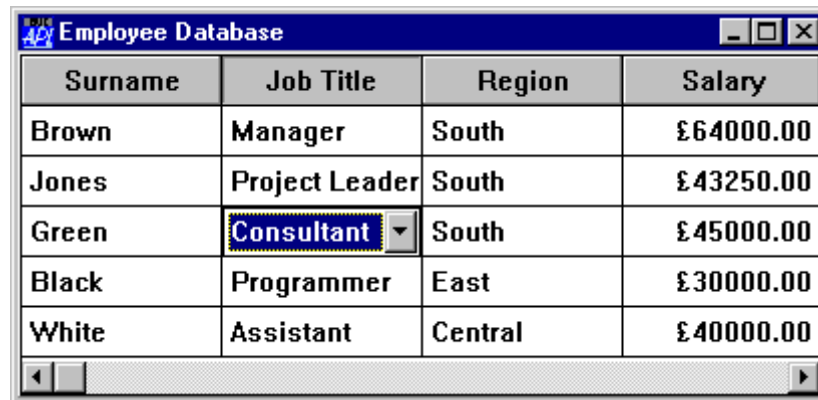
Using Combo Objects

A Combo object is used to present a list of choices for a cell. Although you *may* use an external Combo, internal Combos are more suitable for most applications. If different cells have different sets of choices, you can create several Combo objects, each with its own set of Items and associate different cells with different Combos through the CellTypes property. Alternatively, you can use a single Combo and change Items dynamically from a callback on the CellMove event. In all cases, the value in the cell corresponds to the Text property of the Combo.

If you use a floating Combo, the appearance of the non-current cells depends upon the value of the ShowInput property. If ShowInput is 0 (the default), the non-current cells are drawn in the standard way as if there were no associated input object. If ShowInput is 1, the non-current cells are given the *appearance* of a Combo, although the system does not actually use Combos to do so. Furthermore, there is a subtle difference in behaviour. If ShowInput is 0, the user must click twice to change a value; once to position the Combo on the new cell and again to drop its list box. If ShowInput is 1, the user may drop the list box with a single click on the cell.

Note that ShowInput may be a scalar that applies to the whole Grid, or a vector whose elements applies to different cells through the CellType property.

The following Grid uses two internal Combo objects for the *Job Title* and *Region* columns, but with ShowInput set to 0. Only the current cell has Combo appearance.

Surname	Job Title	Region	Salary
Brown	Manager	South	£64000.00
Jones	Project Leader	South	£43250.00
Green	Consultant ▾	South	£45000.00
Black	Programmer	East	£30000.00
White	Assistant	Central	£40000.00

The same Grid with ShowInput set to 1 is illustrated below. In this case, all of the cells associated with Combo objects have Combo appearance.

Surname	Job Title	Region	Salary
Brown	Manager ▾	South ▾	£64000.00
Jones	Project Lead ▾	South ▾	£43250.00
Green	Consultant ▾	South ▾	£45000.00
Black	Programmer ▾	East ▾	£30000.00
White	Assistant ▾	Central ▾	£40000.00

Using Radio and Check Button Objects

Radio and Check Buttons behave in a similar way to Combo objects except that they may only be used internally. The value in the cell associated with the Button must be 0 or 1 and corresponds to the Button's State property. The value is toggled by clicking the Button.

If ShowInput is 0, the user must click twice to change a value; once to position the (floating) Button on the cell, and a second time to toggle its state. If ShowInput is 1, the user may change cell values directly with a single click. Note that this may be undesirable in certain applications because the user cannot click on a cell without changing its value.

By default, the value of the EdgeStyle property for a Radio or Check Button which is created as the child of a Grid is `'None'`, so you must set EdgeStyle explicitly to `'Plinth'` if a 3-dimensional appearance is required.

You can refine the appearance of the Radio or Check Button using its Align property. This may be set to `'Left'`, `'Right'` or `'Centre'` (and `'Center'`). The latter causes the symbol part of the Button (the circle or checkbox) to be centred within the corresponding Grid cell(s) but should only be used if the Caption property is empty.

The following illustrates different values for the Align property using Check Buttons.

```
'F'⎕WC'Form' 'Aligned Check Boxes in a Grid'

'F.G'⎕WC'Grid' (¯1+?10 3ρ2)(0 0)(100 100)('ShowInput' 1)

'F.G.C1'⎕WC'Button' 'Left' ('Style' 'Check')
         ('EdgeStyle' 'Plinth') ('Align' 'Left')

'F.G.C2'⎕WC'Button' '' ('Style' 'Check')
         ('EdgeStyle' 'Plinth') ('Align' 'Centre')

'F.G.C3'⎕WC'Button' 'Right' ('Style' 'Check')
         ('EdgeStyle' 'Plinth') ('Align' 'Right')

'F.G'⎕WS'Input' ('F.G.C1' 'F.G.C2' 'F.G.C3')

'F.G'⎕WS'CellTypes' (10 3ρ1 2 3)
```

Specifying Individual Cell Attributes

The FCol, BCol, CellFonts and Input properties can be used to specify attributes of individual cells. One possible design would be for these properties to be matrices like the Values property, each of whose elements corresponded to a cell in the Grid. However, although conceptually simple, this design was considered to be wasteful in terms of workspace, especially as it is unlikely that every cell will require a totally individual set of attributes. Instead, FCol, BCol, CellFonts and Input either specify a single attribute to be applied to all cells, or they specify a vector of attributes which are indexed through the CellTypes property. This design is slightly more complex to use, but minimises the workspace needed to represent cell information.

CellTypes is an integer matrix of the same size as Values. Each number in CellTypes defines the *type* of the corresponding cell, where *type* means a particular set of cell attributes defined by the BCol, FCol, CellFonts and Input properties.

If an element of CellTypes is 0 or 1, the corresponding cell is displayed using the *normal* value of each of the FCol, BCol, CellFonts and Input properties. The normal value is either the value defined by its first element or, if the property has not been specified, its default value.

If an element of CellTypes is greater than 1, the corresponding element of each of the FCol, BCol, CellFonts and Input properties is used. However, if a particular property applies to all cells, you need only specify one value; there is no need to repeat it. This mechanism is perhaps best explained by using examples.

Example 1

Suppose that you want to use a Grid to display a numeric matrix DATA and you want to show elements whose value exceeds 150 with a dark grey background. Effectively, there are 2 different types of cell; normal white ones and dark grey ones. This can be achieved as follows:

```
DATA ← ?12 3ρ300

'Test' ⎕WC 'Form' 'Example 1'
'Test.G' ⎕WC 'Grid' DATA (0 0) (100 100)

'Test.G' ⎕WS 'CellTypes' (1+DATA>150)

'Test.G' ⎕WS 'BCol' (255 255 255)(128 128 128)
```

Example 1			
	A	**B**	**C**
1	109	234	216
2	200	175	106
3	48	88	51
4	68	202	122
5	266	174	103
6	132	210	251
7	238	255	85
8	80	51	161
9	22	300	226
10	37	50	128
11	232	65	131

CellTypes[3;3] = 1, so cell uses first element of Bcol which is 255 255 255 (white)

CellTypes[6;3] = 2, so cell uses second element of Bcol which is 128 128 128 (dark grey)

Example 2

Continuing on from the first example, suppose that in addition, you want to show values
that exceed 200 with a white background, but using a bold font. Now you have 3 types
of cell; white background with normal font, dark grey background with normal font,
and white background with bold font. This can be done as follows:

```
CT ←(DATA > 200) + 1+DATA > 100
'Test.G' ⎕WS 'CellTypes' CT

COL ← (255 255 255) (128 128 128) (255 255 255)
'Test.G' ⎕WS 'BCol' COL

'Normal' ⎕WC 'Font' 'Helvetica' 16
'Bold'   ⎕WC 'Font' 'Helvetica' 16 ('Weight' 1000)

'Test.G' ⎕WS 'CellFonts' 'Normal' 'Normal' 'Bold'
```

Example 2			
	A	**B**	**C**
1	109	**234**	**216**
2	200	175	106
3	48	88	51
4	68	**202**	122
5	**266**	174	103
6	132	**210**	**251**
7	**238**	**255**	85
8	80	51	161
9	22	**300**	**226**
10	37	50	128
11	**232**	65	131

CellTypes[2;3]=1,
so cell uses BCol[1] which
is 255 255 255 (white) and
CellFonts[1] which is
'Normal'.

CellTypes[5;2]=2, so cell
uses BCol[2] which is
128 128 128 (dark grey)
and CellFonts[2] which
is 'Normal'

CellTypes[9;2]=3, so cell
uses Bcol[3] which is
255 255 255 (white) and
CellFonts[3] which is
'Bold'

Example 3

This is a more complex example that introduces different uses of the Input property to handle numeric and date cells. Suppose that you wish to display the names, date of birth, and salaries of some people. The user may edit the salary and date of birth, but not the name. Salaries in excess of $19,999 are to be shown in bold

This means that we need 4 types of cell; the "names" cells, the "date of birth" cells, the cells containing salaries below $20,000 and those cells containing $20,000 or more. The Input property must specify 3 different objects; a Label for the protected "names" cells, an Edit object for the "date" cells, and a different Edit object for the salaries. The CellFonts property must specify the two different fonts required; normal and bold.

```
'Test.G' ⎕WS 'Values' ([0.5] NAMES BIRTHDATES SALARIES)

CT ←1,2,[1.5]3+SALARIES > 19999
'Test.G' ⎕WS 'CellTypes' CT

'Test.G.Name' ⎕WC 'Label'
'Test.G.Date' ⎕WC 'Edit' ('FieldType' 'Date')
'Test.G.Sal'  ⎕WC 'Edit' ('FieldType' 'Currency')
INPUTS ← 'Test.G.Name' 'Test.G.Date', 2ρ⊂'Test.G.Sal'
'Test.G' ⎕WS 'Input' INPUTS

'Normal' ⎕WC 'Font' 'Arial' 12
'Bold'   ⎕WC 'Font' 'Arial' 12 ('Weight' 1000)
FONTS ← (3ρ⊂'Normal'), ⊂'Bold'
'Test.G' ⎕WS 'CellFonts' FONTS
```

	Name	Date of Birth	Salary	
1	Jones	08/11/83	$10303.00	
2	Smith	05/03/84	**$28263.00**	
3	White	27/07/82	$10866.00	
4	Baker	06/04/84	$10562.00	
5	O'Donnel	29/10/82	$2420.00	
6	Black	25/10/83	$17157.00	
7	Redman	23/06/84	$23292.00	
8	Green	31/12/82	**$25327.00**	
9	Anderson	21/10/83	$14799.00	
10	Andrews	25/06/82	**$24520.00**	
11	Batchelor	28/02/83	$19283.00	

Celltypes[2;1]=1

CellTypes[4;2]=2

CellTypes[6;3]=3

CellTypes[10:3]=4

Drawing Graphics on a Grid

You may draw graphics on a Grid by creating graphical objects (Circle, Ellipse, Image, Marker, Poly, Rect and text) as *children* of the Grid.

For the Grid (but only for the Grid) the Coord property may be set to `'Cell'` as an alternative to `'Prop'`, `'Pixel'` or `'User'`. This allows you to easily position graphical objects relative to individual cells or ranges of cells. The origin of the Grid (0,0) is deemed to be the top left corner of the data (i.e. the area inside the row and column titles). In Cell co-ordinates, the value (1,1) is therefore the bottom right corner of the first cell. Regardless of the coordinate system, graphical objects scroll with the data.

The following example illustrates how to draw a box around the cells in rows 2 to 4 and columns 3 to 6.

```
'F'⎕WC'Form' 'Graphics on a Grid'
'F.G'⎕WC'Grid' (?10 10ρ100)(0 0)(100 100) ('CellWidths' 10)
'F.G.L'⎕WC'Rect'(1 3)(3 4)('LWidth' 4)('Coord' 'Cell')
```

The OnTop property of the graphical object controls how it is drawn relative to the grid lines and cell text. For graphical objects created as a child of a Grid, OnTop may be 0, 1 or 2.

0	Graphical object is drawn behind grid lines and cell text
1	Graphical object is drawn on top of grid lines but behind cell text
2	Graphical object is drawn on top of grid lines and cell text

The following example shows the effect of the OnTop property on how an Image is drawn on a Grid.

```
'F'⎕WC'Form' 'Graphics on a Grid' ('Size' 45 50)
'F.G'⎕WC'Grid' (?10 10ρ100)(0 0)(100 100) ('CellWidths' 10)
'F.M'⎕WC'MetaFile' 'C:\WDYALOG\WS\DOLLAR'
'F.G.I'⎕WC'Image'(0 0)('Size' 10 10)('Coord' 'Cell')

'F.G.I'⎕WS('Picture' 'F.M')('Ontop' 0)
```

```
'F.G.I'⎕WS 'OnTop' 1
```



```
'F.G.I'⎕WS 'OnTop' 2
```
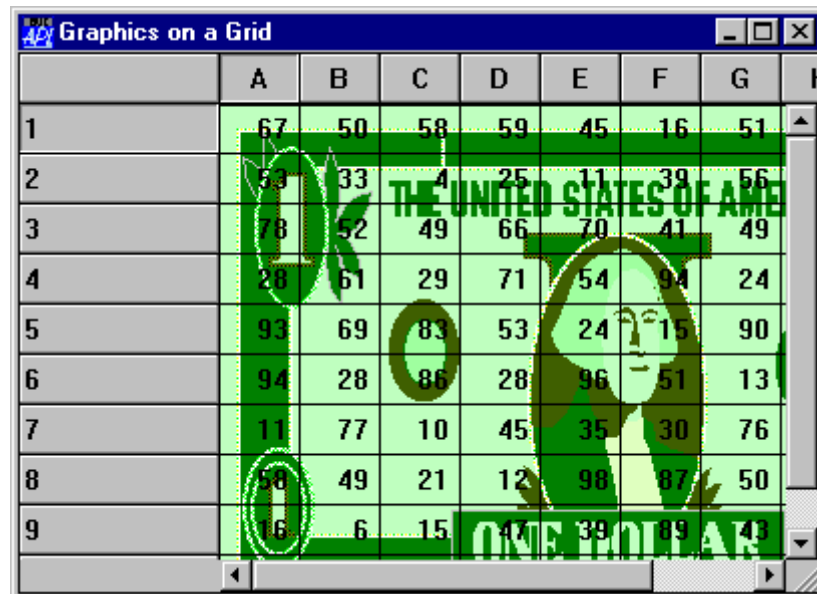
Controlling User Input

The Grid object is designed to allow you to implement simple applications with very little programming effort. You merely present the data to be edited by setting the Values property and then get it back again once the user has signalled completion. The validation imposed by the associated Edit object(s) will prevent the user from entering invalid data and your program can leave the user interaction to be managed entirely by APL. However, for more sophisticated applications, the Grid triggers events which allow your program to respond dynamically to user actions.

Moving from Cell to Cell

When the user moves from one cell to another, the Grid generates a CellMove event. This reports the co-ordinates (row and column) of the newly selected cell. The CellMove event serves two purposes. Firstly, it allows you to take some special action when the user selects a particular cell. For example, you could display a Combo or List object to let the user choose a new value from a pre-defined set, then copy the selected value into the cell. Secondly, the CellMove event provides the means for you to position the user in a particular cell under program control, using ⎕NQ.

Changing Standard Validation Behaviour

Input validation is provided by the Edit object associated with a cell. By default, the built-in validation will prevent the user from leaving the cell should the data in that cell be invalid. For example, if the FieldType is `'Date'` and the user enters 29th February and a non-leap year, APL will beep and not allow the user to leave the cell until a valid date has been entered If you wish instead to take some other action, for example display a message box, you should use the CellError event. This event is generated immediately the user attempts to move to another cell when the data in the current cell is invalid. The event is also generated if the user selects a MenuItem, presses a Button or otherwise changes the focus away from the current cell.

The CellError event reports the row and column number of the current cell, the (invalid) text string in that cell, the name of the object to which the user has transferred attention or the co-ordinates of the new cell selected. The default action of the event is to beep, so to disable the beep your callback function should return a 0. If you wish to allow the user to move to a different cell, you must do so explicitly by generating a CellMove event using ⎕NQ or by returning a CellMove event as the result of the callback.

Reacting to Changes

If enabled, the Grid object generates a CellChange event whenever the user alters data in a cell and then attempts to move to another cell or otherwise shifts the focus away from the current cell. This allows you to perform additional validation or to trigger calculations when the user changes a value. The CellChange event reports the co-ordinates of the current cell and the new value, together with information about the newly selected cell or the external object to which the focus has changed.

The default action of the CellChange event is to replace the current value of the cell with the new one. If you wish to prevent this happening, your callback function must return a 0. If in addition you wish the focus to remain on the current cell, you must do this explicitly by using the CellMove event to reposition the current cell back to the one the user has attempted to leave.

Restoring User Changes

The Grid object supports an Undo method which causes the last change made by the user to be reversed. This method can only be invoked under program control using ⎕NQ and cannot be directly generated by the user. If you want to provide an *undo* facility, it is recommended that you attach a suitable callback function to a MenuItem or a Button. To perform an undo operation, the callback function should then generate an Undo event for the Grid object.

Updating Cell Data

You can change the entire contents of the Grid by resetting its Values property with ⎕WS. However, this will causes the entire Grid to be redrawn and is not to be recommended if you only want to change one cell or just a few cells.

You can change the value in a particular cell by using ⎕NQ to send a CellChange event to the Grid. For example, if you want to alter the value in row 2 column 3 of the Grid object called `Test.G` to 42, you simply execute the following statement :

```
⎕NQ 'Test.G' 'CellChange' 2 3 42
```

To update an entire row or column of data you can use the RowChange and ColChange events. For example, to change all 12 columns of row 500 to the 12-element vector `TOTAL`, you could execute :

```
⎕NQ 'Test.G' 'RowChange' 500 TOTAL
```

Deleting Rows and Columns

You can delete a row or column by using ⎕NQ to send a DelRow or DelCol message to the Grid object. For example, the following statement deletes the 123rd row from the Grid object `Test.G`. Note that if you have specified it, the corresponding element of RowTitles is removed too.

```
⎕NQ 'Test.G' 'DelRow' 123
```

Inserting Rows and Columns

You can insert or add a row or column using the AddRow or AddCol method. You must specify the following information.

> row or column number
> title (optional)
> height or width (optional)
> undo flag (optional)
> resize flag (optional)
> title colour (optional)
> gridline type (optional)

The event message must specify the number of the row or column you wish to insert. This is index-origin dependent and indicates the number that the row or column will have after it has been inserted. For example, if ⎕IO is 1 and you wish to insert a row between the 10th and 11th rows, you specify the number of the row to be inserted as 11. If you wish to insert a new column before the first one, you specify a column number of 1. To append a row or column to the end of the Grid, you should specify 1 + the current number of rows or columns.

If you have specified RowTitles or ColTitles, the message may include a title for the new row or column and this will be inserted in RowTitles or ColTitles as appropriate. If you fail to supply a new title, an empty vector will be inserted in RowTitles or ColTitles for you. If you are using default row and column headers and you have not specified RowTitles or ColTitles, any title you supply will be ignored. In this case the rows and columns will be re-labelled automatically.

If you have set CellHeights or CellWidths to a vector, the AddRow or AddCol event message may include the height or width of the new row or column being inserted. If you fail to supply one or you specify a value of ¯1 the default value will apply. Note that setting the height or width to 0 *is* allowed and will cause the new row or column to be invisible. If CellHeights or CellWidths has not been specified or is a scalar, the new row or column will be given the same height or width as the others and any value that you specify is ignored.

The undo flag indicates whether or not the insertion will be added to the undo stack and may therefore be subsequently undone. Its default value is 1.

If the data in the Grid is entirely numeric, the new row or column will be filled with zeros. If not, it will be filled with empty character vectors. If you want to set the row or column data explicitly, you should invoke the ChangeRow or ChangeCol immediately after the AddRow or AddCol event. The ChangeRow and ChangeCol event require just the row or column number followed by the new data.

The following example adds a new row entitled "Chateau Latour" to a Grid object called `Test.G`. The first statement adds a new row between rows 122 and 123 (it becomes row 123) of the Grid. It will be of default height (or the same as all the other rows) and the change may not be undone (the undo flag is 0). The second statement sets the data in the new row to the values defined by the vector `LATOUR_SALES`.

```
⎕NQ 'Test.G' 'AddRow' 123 'Chateau Latour' ¯1 0
⎕NQ 'Test.G' 'ChangeRow' 123 LATOUR_SALES
```

TreeView Feature

Introduction
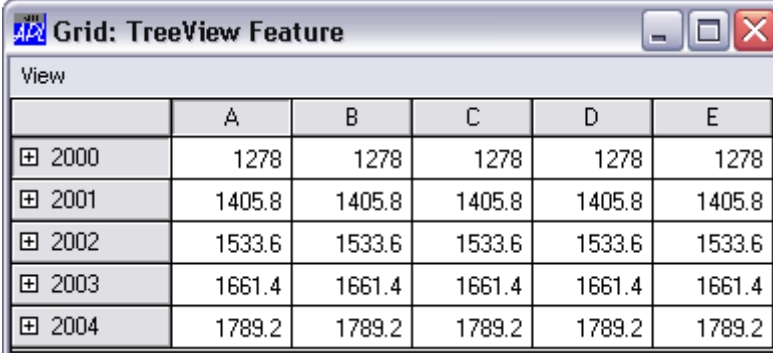
The Grid can display a *TreeView like* interface in the row titles and automatically shows and hides row of data as the user expands and contracts nodes of the tree.

RowTreeDepth property

The tree structure is specified by the RowTreeDepth property This is either a scalar 0 or an integer vector of the same length as the number of rows in the grid. RowTreeDepth is similar to the Depth property of the TreeView object.

Each element of RowTreeDepth specifies the depth of the corresponding row of the Grid. A value of 0 indicates that the row is a top-level row. A value of 1 indicates that the corresponding row is a child of the most recent row whose RowTreeDepth is 0; a value of 2 indicates that the corresponding row is a child of the most recent row whose RowTreeDepth is 1, and so forth.

The picture below illustrates the initial appearance of a Grid with TreeView behaviour. Notice that at first only the top-level rows are displayed.

The `TreeGrid` function that was used to generate this Grid is shown overleaf. The tree structure is defined on `TreeGrid[26]`.
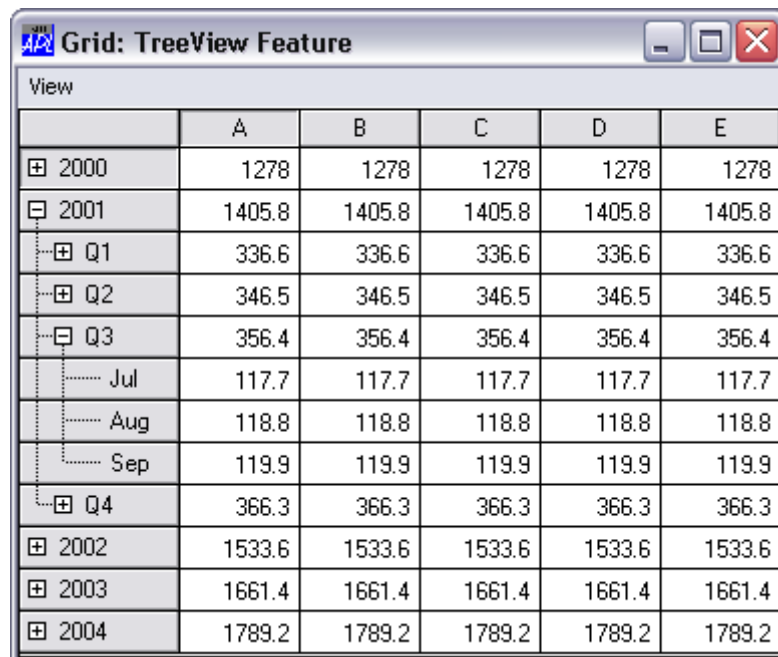
In this example, the Grid has top-level rows (RowTreeDepth of 0) that contain annual totals. The second-tier rows (RowTreeDepth of 1), contain quarterly totals, while the third-tier rows (RowTreeDepth of 2) contain monthly figures.

```
      ∇ TreeGrid;SIZE;YR;YRS;DATA;MDATA;QDATA;YDATA;IX
[1]    SIZE←380 400
[2]    'F'⎕WC'Form' 'Grid: TreeView Feature'
                  ('Coord' 'Pixel')
[3]    F.Size←SIZE
[4]    'F.MB'⎕WC'MenuBar'
[5]    'F.MB.View'⎕WC'Menu' 'View'
[6]    'F.MB.View.Expand1'⎕WC'MenuItem' 'Expand Years'
[7]    'F.MB.View.Expand1'⎕WS'Event' 'Select'
                              '≜F.G.RowSetVisibleDepth 1'
[8]    'F.MB.View.Expand2'⎕WC'MenuItem' 'Expand All'
[9]    'F.MB.View.Expand2'⎕WS'Event' 'Select'
                              '≜F.G.RowSetVisibleDepth 2'
[10]   'F.MB.View.Collapse'⎕WC'MenuItem' 'Collapse All'
[11]   'F.MB.View.Collapse'⎕WS'Event' 'Select'
                              '≜F.G.RowSetVisibleDepth 0'
[12]   'F.G'⎕WC'Grid'('Posn' 0 0)SIZE
[13]   F.G.(TitleWidth CellWidths←80 60)
[14]   YR←'Q1' 'Jan' 'Feb' 'Mar' 'Q2' 'Apr' 'May' 'Jun'
[15]   YR,←'Q3' 'Jul' 'Aug' 'Sep' 'Q4' 'Oct' 'Nov' 'Dec'
[16]   YRS←'2000' '2001' '2002' '2003' '2004'
[17]   F.G.RowTitles←⊃,/(⊂¨YRS),¨⊂YR
[18]   MDATA←12 5ρ5/100+ι12
[19]   YDATA←+⌿MDATA
[20]   QDATA←(3+/[1]MDATA)[1 4 7 10;]
[21]   MDATA←((ρYR)ρ0 1 1 1)\MDATA
[22]   MDATA[1 5 9 13;]←QDATA
[23]   YDATA←YDATA,[1]MDATA
[24]   DATA←⊃,[1]/1 1.1 1.2 1.3 1.4×⊂YDATA
[25]   F.G.Values←DATA
[26]   F.G.RowTreeDepth←(ρF.G.RowTitles)ρ0,(ρYR)ρ1 2 2 2
      ∇
```

When the user clicks on one of the nodes indicated by a "+" symbol, the Grid automatically expands to display the rows at the next level below that node. At the same time, an Expanding event is generated. In the next picture, the user has clicked on the *2001* node and, below that, the *Q3* node.
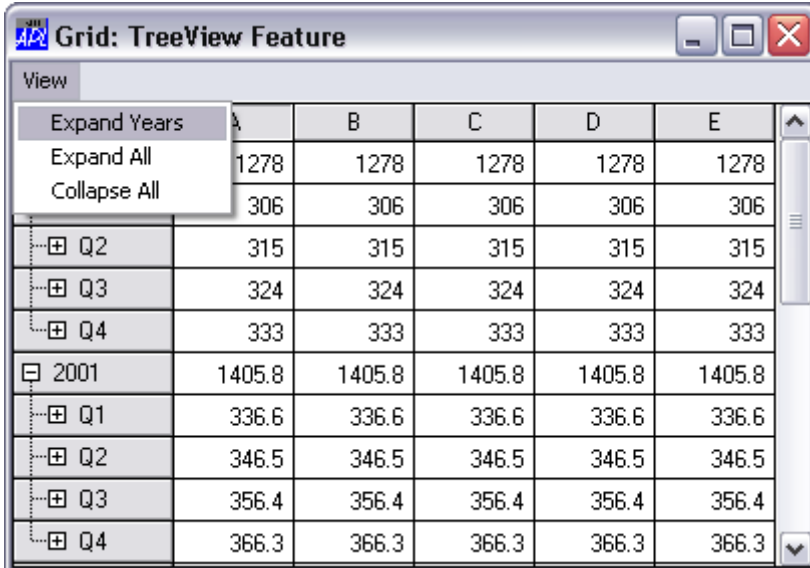
	A	B	C	D	E
⊞ 2000	1278	1278	1278	1278	1278
⊟ 2001	1405.8	1405.8	1405.8	1405.8	1405.8
⊞ Q1	336.6	336.6	336.6	336.6	336.6
⊞ Q2	346.5	346.5	346.5	346.5	346.5
⊟ Q3	356.4	356.4	356.4	356.4	356.4
Jul	117.7	117.7	117.7	117.7	117.7
Aug	118.8	118.8	118.8	118.8	118.8
Sep	119.9	119.9	119.9	119.9	119.9
⊞ Q4	366.3	366.3	366.3	366.3	366.3
⊞ 2002	1533.6	1533.6	1533.6	1533.6	1533.6
⊞ 2003	1661.4	1661.4	1661.4	1661.4	1661.4
⊞ 2004	1789.2	1789.2	1789.2	1789.2	1789.2

RowSetVisibleDepth Method

The Grid provides a RowSetVisibleDepth method that provides *tier-level* control over which rows are displayed.
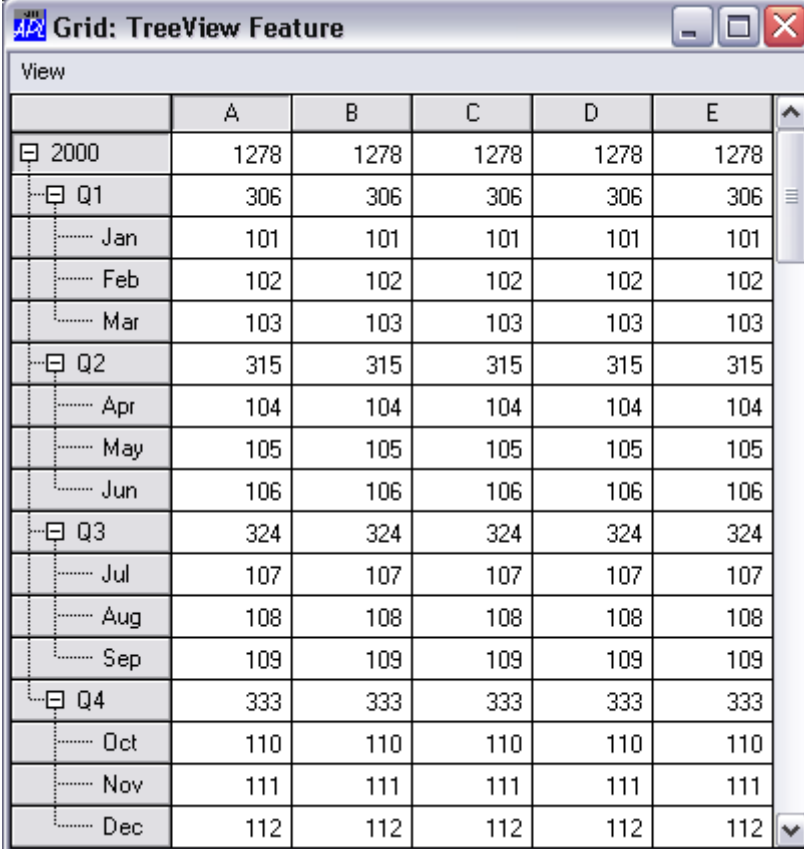
The value of its argument is an integer that specifies the depth of rows to be displayed. The Grid displays all rows whose RowTreeDepth values are less than or equal to this value. In the example, this method is called by items on the *View* menu.

The next picture shows how the Grid is displayed after choosing *Expand Years* from the *View* menu. Notice that, as specified by `TreeGrid[6]` this menu item simply executes the RowSetVisibleDepth method with an argument of 1.

Similarly, the *Expand All item executes* RowSetVisibleDepth 2, as specified by `TreeGrid[7]` and this causes the Grid to display all rows up to and including RowTreeDepth of 2 as shown below.

	A	B	C	D	E
⊟ 2000	1278	1278	1278	1278	1278
⊟ Q1	306	306	306	306	306
Jan	101	101	101	101	101
Feb	102	102	102	102	102
Mar	103	103	103	103	103
⊟ Q2	315	315	315	315	315
Apr	104	104	104	104	104
May	105	105	105	105	105
Jun	106	106	106	106	106
⊟ Q3	324	324	324	324	324
Jul	107	107	107	107	107
Aug	108	108	108	108	108
Sep	109	109	109	109	109
⊟ Q4	333	333	333	333	333
Oct	110	110	110	110	110
Nov	111	111	111	111	111
Dec	112	112	112	112	112

Note that the *Collapse All* item executes RowSetVisibleDepth 0, which causes only the top-level rows to be displayed.

You may open specific nodes by invoking the Expanding event as a method.

Fine control over the appearance of the tree is provided through the RowTreeImages and RowTreeStyle properties. See Object Reference for further details.

Grid Comments

Introduction

Grid comments are implemented in a manner that is consistent with the way comments are handled in Microsoft Excel.

If a comment is associated with a cell, a small red triangle is displayed in its top right corner. When the user rests the mouse pointer over a commented cell, the comment is displayed as a pop-up with an arrow pointing back to the cell to which it refers. The comment disappears when the mouse pointer is moved away. This is referred to as *tip* behaviour.

It is also possible to display and hide comments under program control. A comment window displayed under program control does not (normally) disappear automatically when the user moves the mouse, but instead must be hidden explicitly. It is therefore possible to have several comments visible.

Implementation

Because comments are typically sparse, this facility is implemented by a small set of *methods* rather than as a property, and comments are stored internally in data structures that minimise storage space. The following methods and events are provided.

Event/Method	Number	Description
AddComment	220	Associates a comment with a cell
DelComment	221	Deletes the comment associated with a particular cell
GetComment	222	Retrieves the comment associated with a given cell
ShowComment	223	Displays a comment either as a pop-up or on-top window
HideComment	224	Hides a comment
ClickComment	225	Reported when user clicks the mouse on a comment window

A comment is described by its text content and the size of the window in which it appears. The text may optionally be *Rich Text* (RTF) such as that produced by the value of the RTFText property of a RichEdit object. The size of the window is specified in pixels.

AddComment Method

This method is used to add a new comment. For example, the following statement associates a comment with the cell at row 2, column 1; the text of the comment is "Hello", and the size of the comment window is 50 pixels (high) by 60 pixels (wide).

```
    2 ⎕NQ'F.G' 'AddComment' 2 1 'Hello' 50 60
```

The height and width of the comment window, specified by the last 2 elements of the right argument to ⎕NQ are both optional. If the cell already has an associated comment, the new comment replaces it.

Note that just before the comment is displayed, the Grid generates a ShowComment event which gives you the opportunity to (temporarily) change the text and/or window size of a comment dynamically.

DelComment Method

This method is used to delete a comment. For example, the following expression removes the comment associated with the cell at row 2, column 1.

```
    2 ⎕NQ'F.G' 'DelComment' 2 1
```

If the row and column number are omitted, all comments are deleted.

GetComment Method

This method is used to retrieve the comment associated with a cell. For example, the following expression retrieves the comment associated with the cell at row 3, column 1.

```
      ⎕←2 ⎕NQ 'F.G' 'GetComment' 3 1
 1 3  Hello  175 100
```

If there is no comment associated with the specified cell, the result is a scalar 1.

ShowComment Event/Method

If enabled, a Grid will generate a ShowComment event when the user rests the mouse pointer over a commented cell. You may use this event to modify the appearance of the comment dynamically.

You may display the comment associated with a particular cell under program control by generating a ShowComment event using ⎕NQ. By default, a comment displayed under program control does not exhibit tip behaviour but remains visible until it is explicitly removed using the HideComment method.

Note that a comment will only be displayed if the specified cell is marked as a commented cell.

HideComment Event/Method

If enabled, a HideComment event is generated just before a comment window is hidden as a result of the user moving the mouse-pointer away from a commented cell.

Invoked as a method, HideComment is used to hide a comment that has previously been displayed by ShowComment. For example, the following expression hides the comment associated with the cell at row 2, column 1.

```
2 ⎕NQ'F.G' 'HideComment' 2 1
```

ClickComment Event

If enabled, a ClickComment event is generated when the user clicks the mouse in a comment widow. The event message reports the co-ordinates of the cell. The result of a callback function (if any) is ignored.

C H A P T E R 7

Multiple-Document (MDI) Applications

The multiple-document interface (MDI) is a document-oriented interface that is commonly used by word-processors, spreadsheets and other applications that deal with *documents*. An MDI application allows the user to display multiple documents at the same time, with each document displayed in its own window. Document windows are implemented as child forms that are contained within a parent form. When a child form is minimised, its icon appears on the parent form instead of on the desktop. An example MDI application is illustrated in Figure 8.1.
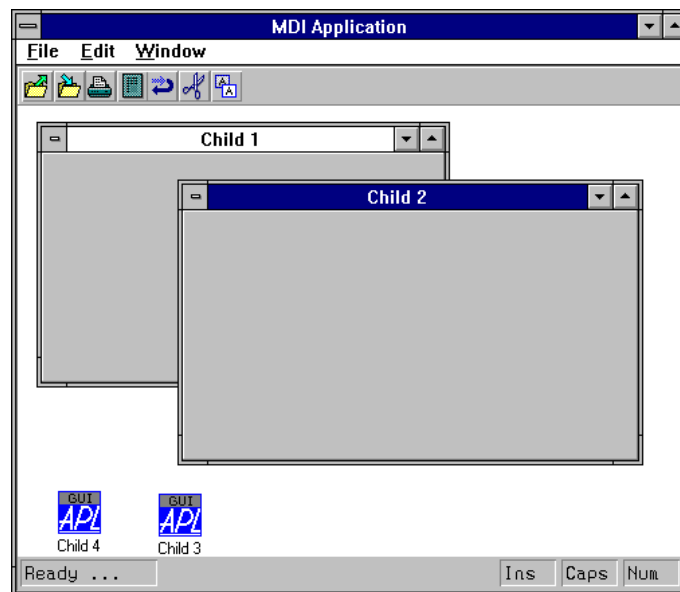
Figure 8.1   Child forms displayed within an MDIClient

In general, the parent form in an MDI application may also contain tool bars and status bars and potentially other objects. This means that not all of the internal area of the parent form is available. To allow for this and to distinguish MDI behaviour from that of simple child forms, Dyalog APL/W uses an MDIClient object.

The MDIClient object is a container object that effectively specifies the client area within the parent Form in which the SubForms are displayed. The MDIClient object also imposes special MDI behaviour which is quite different from that where a SubForm is simply the child of another Form.

By default, the MDIClient occupies the entire client area within its parent Form. This is the area within the Form that is not occupied by ToolBars and StatusBars. In most applications it is therefore not necessary to specify its Posn and Size properties, although you may do so if you want to reserve additional space in the parent Form for other objects.

To Create an MDI Application

1. Create a Form (this will be the parent form for the application).

2. Add MenuBar, ToolBar and StatusBar objects as appropriate as children of the parent Form.

3. Create an MDIClient object as a child of the parent Form.

4. Create the application's SubForms as children of the MDIClient, not as children of the parent Form.

MDI Behaviour

- All child forms are displayed within the MDIClient. Forms may be moved and resized but they are restricted to the MDIClient and will be clipped if they extend beyond it.

- When a child form is minimised, its icon appears on the MDIClient rather than on the desktop.

- When a SubForm is maximised, its Caption is combined with the Caption of the parent Form, i.e. the parent of the MDIClient object and is displayed in the parent Form's title bar. In addition, the SubForm's system menu and restore button are displayed in the parent Form's MenuBar.

- You cannot hide a SubForm. Setting its Visible property to 0 has no effect.

- A SubForm does not display its MenuBar. Instead, it is displayed in place of the parent Form's MenuBar when the SubForm has the focus.
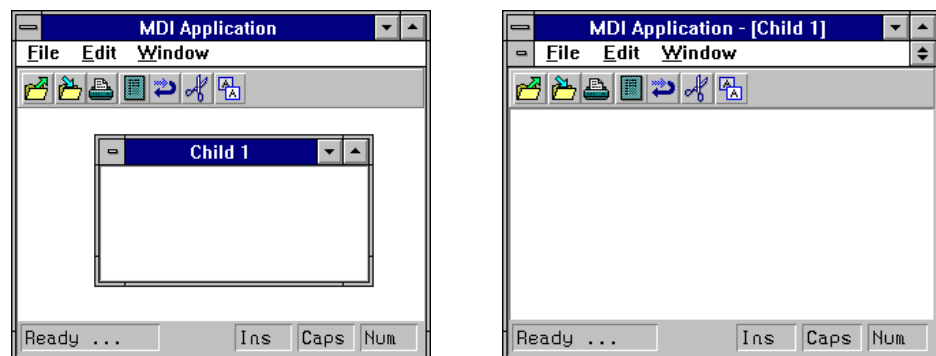
Figure 8.2   The effect of maximising a SubForm

Menus in MDI Applications

A feature of MDI behaviour is that SubForms do not display menu bars. However, if you create a MenuBar object for a SubForm, that object will be displayed as the menu bar of the parent Form whenever the SubForm has the focus. If there are no SubForms or if the SubForm with the focus does not own a MenuBar, the MenuBar of the parent Form is displayed. This mechanism provides one way of achieving the desired effect, namely that the menu bar displayed is appropriate for the type of document represented by the SubForm that has the focus. However, if you have a large number of SubForms of the same type (i.e. which share the same menu bar) you must defined identical MenuBar objects for all of them.

An alternative approach is to define separate MenuBar objects as children of the parent Form, only one of which is visible. Then you simply attach a callback function to the GotFocus event for each SubForm that makes the appropriate MenuBar visible. This approach means that you need only define MenuBar objects for each different *type* of SubForm, rather than for every one.

It is possible to mix these techniques, so that the MenuBar displayed is either the result of your callback function making it visible, or because a SubForm has its own MenuBar object defined and received the focus.

Note that when the user maximises a SubForm, its system menu button and restore button are displayed in the parent Form's menu bar. It is therefore essential that you ensure that your application provides such a menu bar at all times. Otherwise, when your user maximises a SubForm there is no way to reverse it.

Defining a Window Menu

Most MDI applications incorporate a Window menu. This is a special menu that displays the captions of all open SubForms as shown in Figure 8.3. The caption of the SubForm which currently has the focus is checked and the user can switch focus to another SubForm by selecting it from the Window menu.
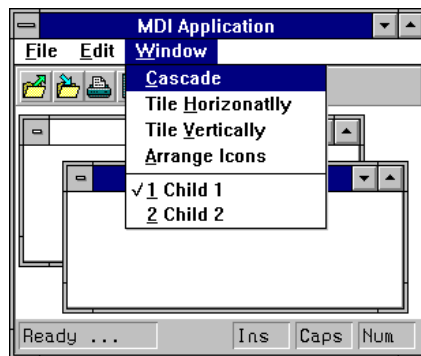
Figure 8.3   The Window menu

The task of updating the Window menu with the names of the SubForms is performed for you by Dyalog APL/W. You nominate the menu to be used for this purpose by setting the MDIMenu property of the appropriate MenuBar object. For example, if your MenuBar is called `F1.MB` and the menu you want to use as the Window menu is called `F1.MB.WM`, you would type the following:

```
    'F1.MB' ⎕WS 'MDIMenu' 'WM'
```

Notice that the name you specify is just the name of the menu itself, not its full pathname. If you have several MenuBars in your application, you must specify the MDIMenu property separately for each one.

Arranging Child Forms and Icons

Another common feature of MDI applications is that the user can ask for the SubForms to be displayed in a particular way, or that any SubForm icons are arranged in an orderly fashion. This is implemented in Dyalog APL/W by your application invoking an method using ⎕NQ. The MDIClient recognises three different methods, namely MDICascade (110), MDITile (111) and MDIArrange (112).

The MDICascade method causes the child forms to be arranged in an overlapping manner. The MDITile method causes them to be tiled, either horizontally or vertically . Finally, the MDIArrange method arranges any child form icons in an orderly fashion. The most convenient way to provide these actions is to attach a Callback function to appropriate MenuItems. The callback function is called with different left arguments according to the MenuItem selected. The following code snippet illustrates this technique.

The following lines define callbacks for each of the MenuItem objects in the Menu `F1.MB.WM`. Each one uses the callback function `MDI_ARRANGE`, but with a left argument corresponding to the message that must be sent to the MDIClient to cause the desired action. For example, clicking the MenuItem named `F1.MB.WM.Vert` runs `MDI_ARRANGE` with a left argument of `(111 1)`

```
'F1.MB.WM.CASCADE' ⎕WS 'Event' 30 'MDI_ARRANGE' 110
'F1.MB.WM.HORZ'    ⎕WS 'Event' 30 'MDI_ARRANGE' (111 0)
'F1.MB.WM.VERT'    ⎕WS 'Event' 30 'MDI_ARRANGE' (111 1)
'F1.MB.WM.ARRANGE' ⎕WS 'Event' 30 'MDI_ARRANGE' 112
```

The `MDI_ARRANGE` function uses its left argument to construct a message for the MDIClient object, in this case `F1.MDI`, and returns it as a result. This causes the desired action.

```
    ∇ MSG←M MDI_ARRANGE MSG
[1]   MSG←(⊂'F1.MDI'),M
    ∇
```

An alternative approach which does not require a callback function is to use ⎕NQ

```
'F1.MB.WM.CASCADE' ⎕WS 'Event' 30 '⍎⎕NQ ''F1.MDI 110'''
'F1.MB.WM.HORZ'    ⎕WS 'Event' 30 '⍎⎕NQ ''F1.MDI 111 0'''
'F1.MB.WM.VERT'    ⎕WS 'Event' 30 '⍎⎕NQ ''F1.MDI 111 1'''
'F1.MB.WM.ARRANGE' ⎕WS 'Event' 30 '⍎⎕NQ ''F1.MDI 112'''
```

C H A P T E R 8

Docking

Introduction

Dyalog APL supports dockable Forms, SubForms, CoolBands and ToolControls.

If an object is dockable, the user may drag it to a different position within the same container, drag it out of its current container and drop it onto a different container, or drop it onto the desktop as a free-floating window. An undocked object can subsequently be redocked in its original container or in another.

For example, a SubForm can be dragged from one Form and docked into another. Or a CoolBand can be dragged out of its CoolBar and turned into a top-level Form on the desktop.

With the exception of ToolControls, when a dockable object is docked or undocked, the full Name and Type of the object change according to the following table.

Dockable Object	Parent Object			
	Form `F1`	SubForm `F1.S1`	CoolBar `F1.CB1`	Root `(.)`
Form `F2`	SubForm `F1.F2`	SubForm `F1.S1.F2`	CoolBand `F1.CB1.F2`	Form `F2`
Form `F1.F2`	SubForm `F1.F2`	SubForm `F1.S1.F2`	CoolBand `F1.CB1.F2`	Form `F1.F2`
SubForm `F2.S2`	SubForm `F1.S2`	SubForm `F1.S1.S2`	CoolBand `F1.CB1.S2`	Form `S2`
CoolBand `F2.CB2.C2`	SubForm `F1.C2`	SubForm `F1.S1.C2`	CoolBand `F1.CB1.C2`	Form `C2`

For example, a top-level Form `F2` when docked in another top-level Form F1, becomes a SubForm named `F2.F1`.

Similarly, a CoolBand named `F2.CB2.C2` when dragged from its CoolBar `F2.CB2` and dropped over the desktop, becomes a top-level Form named `C2`.

Notice how the node name of the object remains the same, but its full pathname changes as it is moved from one parent object to another.

When an object changes Type in this way, the values of all its properties for its original Type are remembered, and these are automatically restored when the object reverts back to its original Type. Since an object can change Type between Form, SubForm, and CoolBand, it follows that there are effectively 3 different sets of properties associated with the object. However, only one set of properties (the set associated with the object's current Type) is visible and accessible (to the programmer) at any one time.

Docking Events

An object (the client) may be docked in another object (the host) if the Dockable property of the client is set to `'Always'` and the name of the client is included in the host object's DockChildren property. This property defines the list of names that the host will accept. Docking a Form or re-docking an already docked object behave in essentially the same way.

DockStart Event

The user picks up a client object by depressing the left mouse button over its title bar or client area and dragging. As soon as the mouse is moved, the object generates a DockStart event At this stage, the entire operation may be cancelled by a callback function on DockStart that returns 0.

Once a docking operation has begun, the outline of the object is displayed as a rectangle that moves with the mouse.

DockMove Event

When the client object is dragged over a suitable host object (one that will accept it as a child), the host object generates a series of DockMove events. Each DockMove event reports the edge along which the client object will be docked, namely Top, Bottom, Left, Right or None, and a corresponding rectangle.

When the mouse pointer approaches an edge of the host, the rectangle changes to describe a docking zone indicating where the object will be docked in the host.

A callback function on DockMove that returns 0 will prevent the outline rectangle changing to indicating a docking zone and will prevent the client from being docked.

A callback function on DockMove can also return a result that modifies the position and size of the rectangle that is actually displayed for the user. This in turn will affect the zone occupied by the client when it becomes docked. For example, you can use this to control its size.

DockRequest Event

When the user releases the mouse pointer, the client object generates a DockRequest event. A callback function on DockRequest may return 0 to abort the operation, or may modify the requested docking zone in the host. In the case of a ToolControl, the callback is used to action the docking operation.

DockAccept Event

In response to a successful DockRequest event, the host object generates a DockAccept event. A callback on DockAccept may also be used to abort the operation or to modify the docking zone. The DockAccept event reports the new name for the client object which it will assume as a child of the host.

Furthermore, if the DockAccept callback actions the event before completing, the docking operation will take place immediately, rather than being deferred until the callback has completed. This allows you to set properties on the newly docked object.

DockEnd Event

Finally, the docked client object (whose name has now changed) will generate a DockEnd event. This is reported for information only and a DockEnd callback function cannot cancel or modify the docking operation. The DockEnd event may however be used to set properties for the newly docked client.

If the user releases the mouse elsewhere than over an accepting host object, the DockEnd event is reported by the client object itself. If appropriate, this will be followed by a Configure event and the client will simply move to a new location without changing its docking status.

DockCancel Event

If at any stage the user presses the Esc key, the operation is aborted and the client object generates a DockCancel event.

Docking a Form inside another

The following example illustrates the effect of docking one Form in another.

```
'Host' ⎕WC 'Form' 'Host'  ('Size' 30 40)
Host.DockChildren←'Client'
```



```
'Client' ⎕WC 'Form' 'Client' ('Size' 20 20)
Client.Dockable←'Always'
```
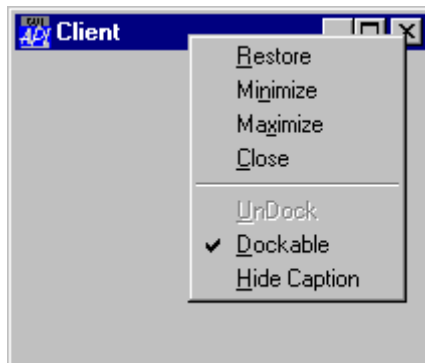

Notice that a dockable Form is indistinguishable in appearance between any other top-level Form except that it has additional items in its pop-up context (right mouse button) menu as shown.

The following picture shows the effect of dragging the `Client` Form to the top edge of the `Host`, just before the mouse button is released.
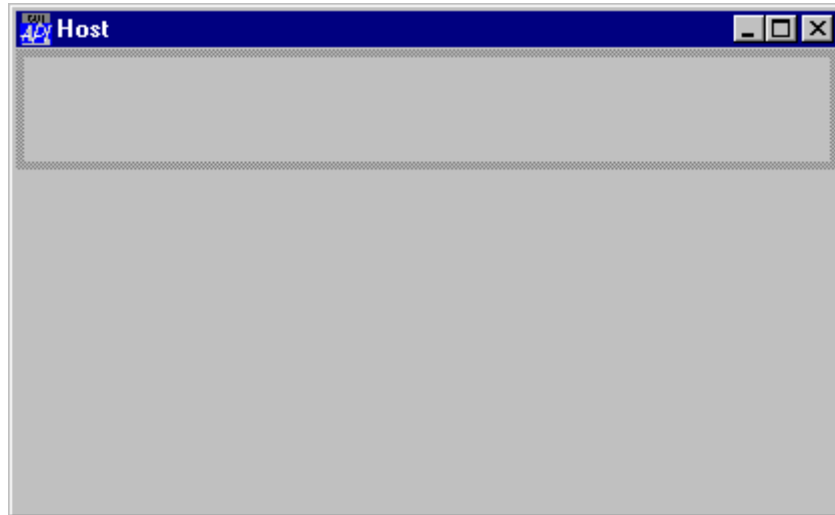
The next picture shows the result after docking. The `Client` Form has become a SubForm (white is the default background colour for a SubForm) called `Host.Client`.
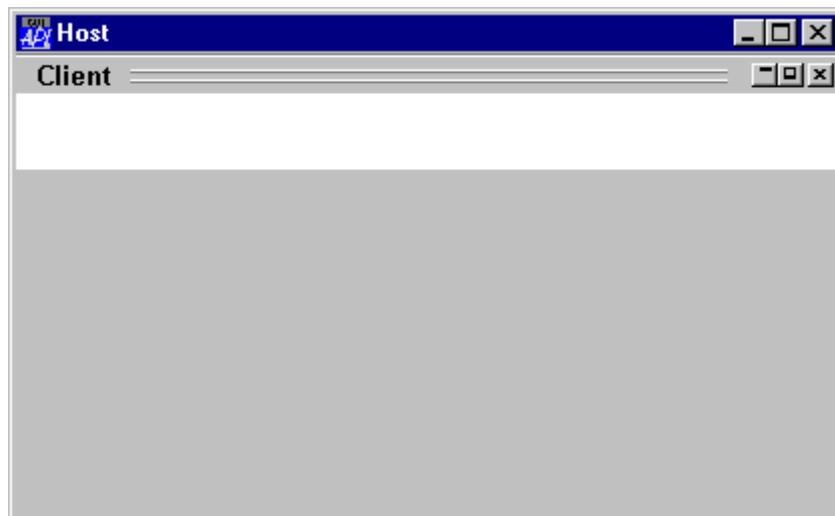
The third picture illustrates the effect of docking the `Client` on the left-hand edge.

The following picture shows the `Client` Form docked as a SubForm along the right edge of the `Host` Form.
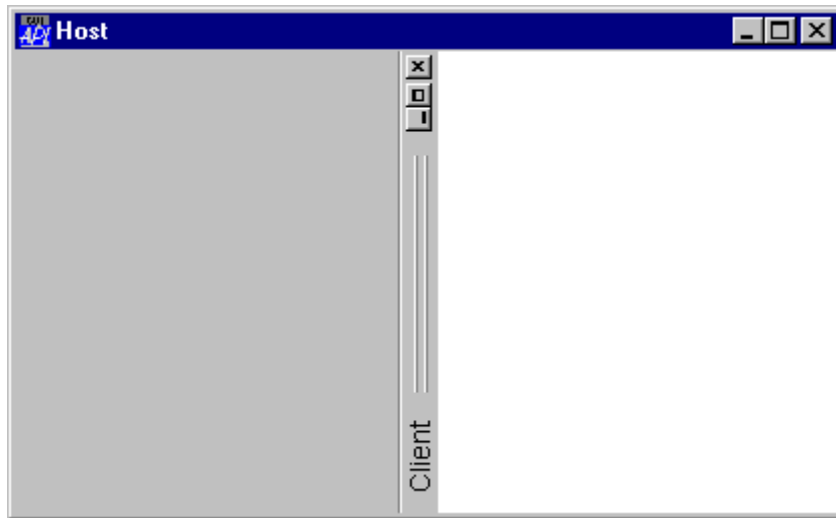
```
'Client2' ⎕WC 'Form' 'Second Client' ('Size' 20 20)
Client2.Dockable←'Always'
```
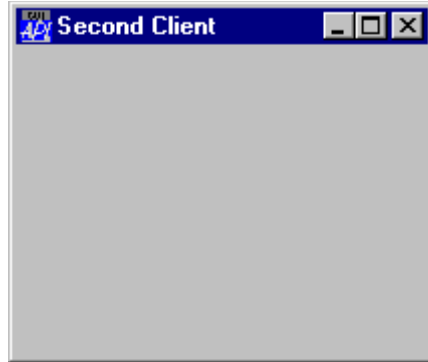

which we can make dockable in both the `Host` Form and the `Host.Client` SubForm:

```
Host.DockChildren Host.Client.DockChildren←'Client2'
```

The next picture shows `Client2` about to be docked in the `Client` SubForm:
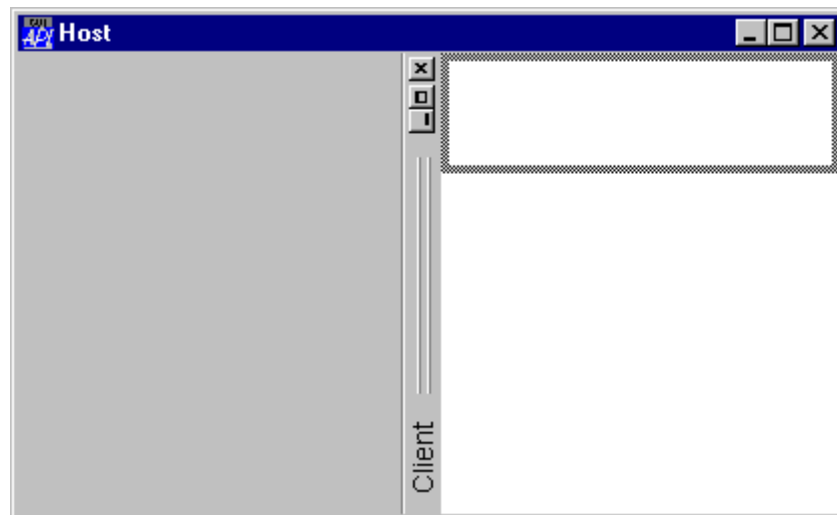
And finally, after it has been docked.
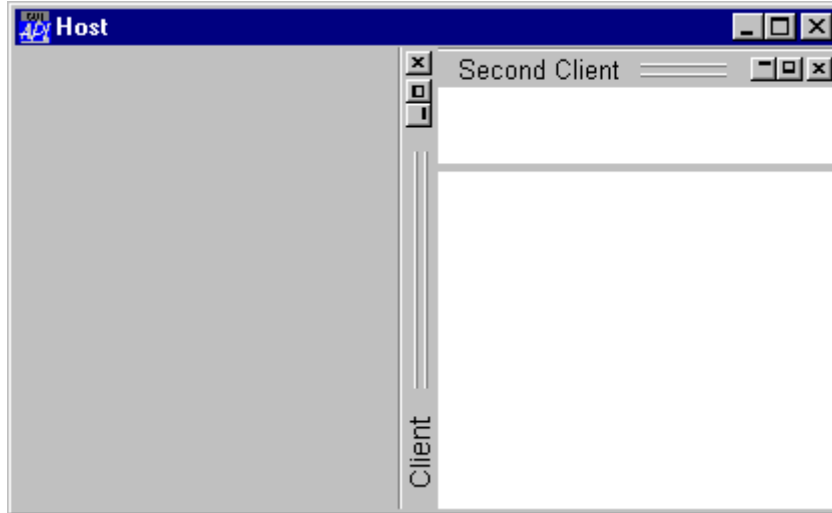
Docking a Form into a CoolBar

The following example illustrates the effect of docking a Form into a CoolBar.

```
      ∇ FormToCoolBand
[1]    'il'⎕WC'ImageList'('Masked' 0)('MapCols' 1)
[2]    'il.'⎕WC'Bitmap'('ComCtl32' 120)⍝ STD_SMALL
[3]
[4]    'host'⎕WC'Form' 'Host'
[5]    host.Coord←'Pixel'
[6]    host.Size←140 350
[7]    'host.cb'⎕WC'CoolBar'
[8]    host.cb.DockChildren←'file' 'edit'
[9]
[10]   :With 'host.cb.file'⎕WC'CoolBand'
[11]       Caption←'File'
[12]       Dockable←'Always'
[13]       'tb'⎕WC'ToolControl'('ImageListObj' '#.il')
[14]       'tb.b1'⎕WC'ToolButton' 'New'('ImageIndex' 7)
[15]       'tb.b2'⎕WC'ToolButton' 'Open'('ImageIndex' 8)
[16]       'tb.b3'⎕WC'ToolButton' 'Save'('ImageIndex' 9)
[17]   :EndWith
[18]
```

```
[19]    :With 'edit'⎕WC'Form' 'Edit'
[20]        Size←100 200
[21]        Dockable←'Always'
[22]        Coord←'Pixel'
[23]        'tb'⎕WC'ToolControl'('ImageListObj' '#.il')
[24]        'tb.b1'⎕WC'ToolButton' 'Cut'('ImageIndex' 1)
[25]        'tb.b2'⎕WC'ToolButton' 'Copy'('ImageIndex' 2)
[26]        'tb.b3'⎕WC'ToolButton' 'Paste'('ImageIndex' 3)
[27]        'tb.b4'⎕WC'ToolButton' 'Undo'('ImageIndex' 4)
[28]        'tb.b5'⎕WC'ToolButton' 'Redo'('ImageIndex' 5)
[29]    :EndWith
        ∇
```
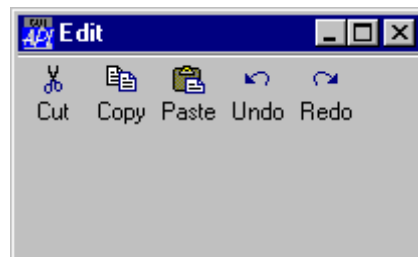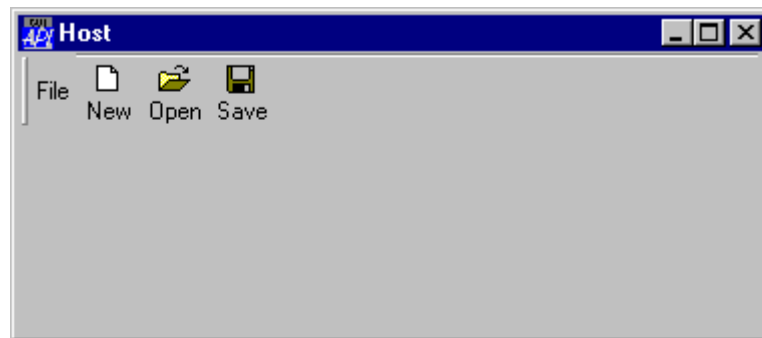

The following picture shows the effect of dragging the client Form to the CoolBar in the host, just before the mouse button is released.
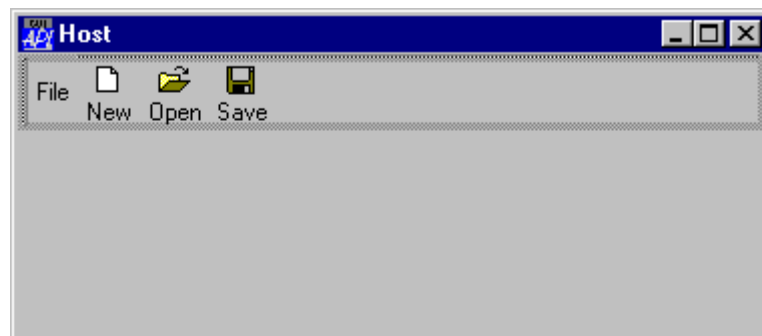
The next picture shows the result after docking. The client Form has become a CoolBand called `host.cb.edit`.
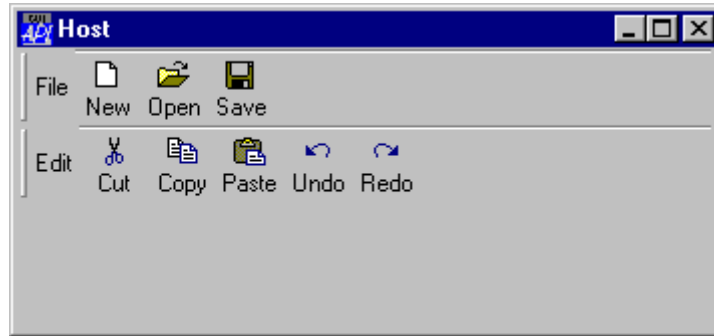
Undocking a SubForm or a CoolBand

When a SubForm or a CoolBand is undocked, it becomes a Form.

The object may either become a Form that is a child of Root, or a Form that remains the child of the Form from where it was undocked. Such an object will always appear *on top of* its parent, even when undocked.

This behaviour is controlled by the UndocksToRoot Property

Note that a Form or a CoolBand object may be undocked if its Dockable property is set to `'Always'`; the DockChildren property does not apply to the Root object.

The Root object does not provide DockMove events, but the docked object will generate a DockRequest event when the user releases the mouse button over the desktop. This may be used to disable or modify the operation.

Docking and Undocking a ToolControl

Docking and undocking a ToolControl is handled rather differently from docking and undocking a Form or CoolBand.

When you undock a ToolControl from a Form or SubForm, it cannot remain a ToolControl object, because a ToolControl cannot be a child of Root. Furthermore, its Type cannot simply change to Form because a Form cannot be a parent of a ToolButton. In fact, a ToolButton may **only** be the child of a ToolControl.

Therefore, when a dockable ToolControl is undocked, no action is taken; you have to perform the various operations yourself.

Typically, you would create a new Form to contain the ToolControl and only the ToolControl), and then delete the original.

The new Form should be dockable in the original parent (of the ToolControl), but a callback should intercept this operation and re-instate the ToolControl as a direct child of the host.

Effectively, when you undock a ToolControl, you need to insert a new (floating) Form between the Host Form and the ToolControl. Then when you re-dock it, you need to remove the (floating) Form from the hierarchy.

The following example illustrates the procedure.

The following function creates a Form containing a dockable ToolControl. The ToolControl can be undocked, becoming a floating toolbar, and then docked back into the original Form.

```
      ∇ DockableToolControl
[1]    'IL'⎕WC'ImageList'('Masked' 0)
[2]    'IL.'⎕WC'Bitmap'('ComCtl32' 120)⍝ STD_SMALL
[3]    :With 'Host'⎕WC'Form' 'Host'
[4]        Coord←'Pixel'
[5]        Size←50 300
[6]        DockChildren←'Floater'
[7]        onDockAccept←'#.DOCK'
[8]        onDockMove←'#.DOCKMOVE'
[9]        :With 'TC'⎕WC'ToolControl'
[10]           Dockable←'Always'
[11]           onDockRequest←'#.UNDOCK'
[12]           ImageListObj←'#.IL'
[13]           'B1'⎕WC'ToolButton' 'New'('ImageIndex' 7)
[14]           'B2'⎕WC'ToolButton' 'Open'('ImageIndex' 8)
[15]           'B3'⎕WC'ToolButton' 'Save'('ImageIndex' 9)
[16]        :EndWith
[17]    :EndWith
      ∇
```

The picture below shows the initial appearance of the `Host` Form and its ToolControl.

Because the ToolControl is dockable, the user may pick it up and drag it out of its parent Form as shown below.
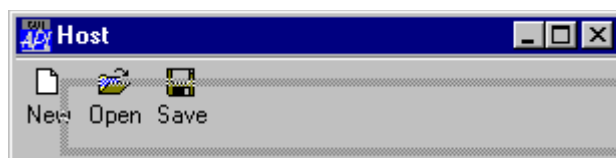
When the user drops the ToolControl outside the **Host** Form, it (the ToolControl) generates a DockRequest event which is attached to the **UNDOCK** callback function. This function, creates a new Form called **Floater**, makes a copy of the ToolControl as a child of **Floater**, and then expunges the original ToolControl from the **Host** Form. The function, and the results of the operation, are shown below. The following points should be noted.

- The **UNDOCK** callback returns 0 to prevent APL from taking any further action (the default action after a successful DockRequest is to generate a DockAccept event, which in this case is undesirable).
- The **Floater** Form is created as a child of the **Host** Form so that it always floats above it in the window stacking order.
- The **Floater** Form is made dockable so that it can be re-docked back into **Host**.
- The (new) ToolControl is made non-dockable, so that the user cannot drag it out of **Floater**.

```
     ∇ R←UNDOCK MSG
[1]    R←0
[2]    :With 'Host.Floater'⎕WC'Form'
[3]        Caption←'Floating ToolControl'
[4]        Dockable←'Always'
[5]        Coord←'Pixel'
[6]        'TC'⎕WC⊃MSG
[7]        TC.Dockable←'Never'
[8]        Size←TC.Size
[9]        Posn←#.Host.Posn+2↑7⊃MSG
[10]   :EndWith
[11]   ⎕EX'#.Host.TC'
     ∇
```


The user may dock the ToolControl back into Host by dragging the **Floater** Form into it.

The `DOCKMOVE` callback function, shown below, prevents the ToolControl (represented by its parent `Floater`) from being docked anywhere except along the top edge.

```
      ∇ R←DOCKMOVE MSG
[1]    ⍝ Only allow docking along Top edge
[2]    R←MSG[4]∊'Top' 'None'
      ∇
```

The picture below illustrates the moment just before the user releases the mouse button to dock Floater back into Host.
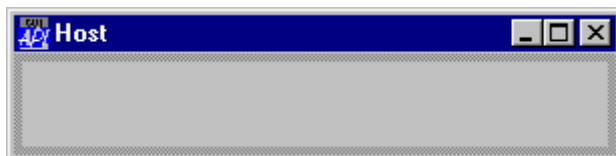
At this point, the `Host` Form generates a DockAccept event and the callback function `DOCK` is invoked. This function recreates the ToolControl as a child of `Host` (making it dockable once more), and then expunges the `Floater` Form.

```
      ∇ R←DOCK MSG
[1]    R←0
[2]    :With ⊃MSG
[3]        'TC'⎕WC ⎕OR(3⊃MSG).TC
[4]        TC.Dockable←'Always'
[5]    :EndWith
[6]    ⎕EX'#.Host.Floater'
      ∇
```

Once again, the result of the callback function is 0 to tell APL that you have dealt with the situation and it is to take no further action.

XP Look and Feel

Windows *XP Look and Feel* is an optional feature of Windows XP and other advanced versions of Windows.

Under XP, it may be enabled from the *Appearance* tab of the *Display Properties* dialog box, by choosing *Windows XP style*.

If XP Look and Feel is enabled, APL will optionally display the title bars of docked windows using the appropriate XP style. You can control this behaviour using the XPLookAndFeelDocker parameter (see User Guide, Chapter 2).

If XPLookAndFeelDocker is 1, APL will display docked window title bars using the appropriate XP style. If XPLookAndFeelDocker is 0 (the default), it will not.

The picture below illustrates how the first example in this chapter appears when XP Look and Feel is enabled, XPLookAndFeelDocker is 1, and a special Windows XP Theme is in use.
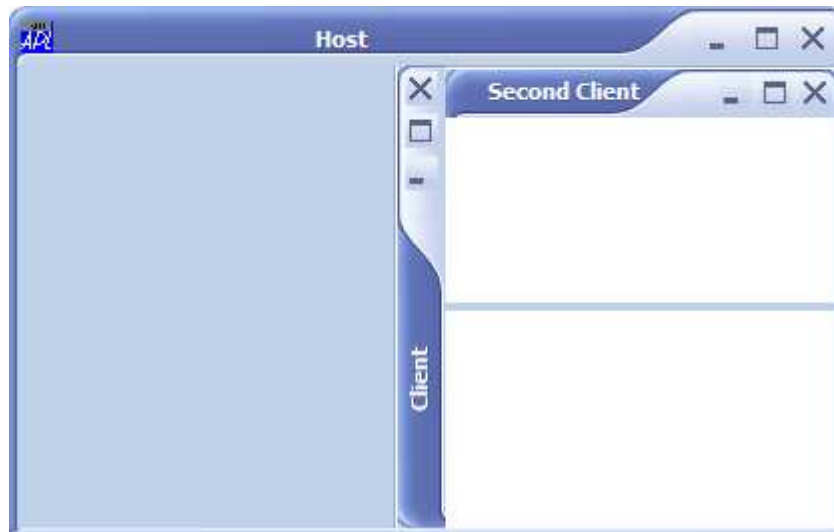
C H A P T E R   9

TCP/IP Support

Introduction

The TCPSocket object provides an event-driven interface to the WinSock network API, which itself is an implementation of TCP/IP for Microsoft Windows.

The TCPSocket object allows you to communicate with other TCP/IP applications running on any computer in your network, including the World Wide Web.

It also provides the mechanism for client/server operation between two Dyalog APL workspaces.

From Version 12.0, a new tool called Conga is the recommended mechanism for connecting to the internet. The code samples in this chapter have not been ported to the Unicode Edition, but continue to work in Classic Editions. For information on accessing the internet and other TCP services in the Unicode Edition, see the *Conga User Guide*.

Two types of TCP/IP connections are possible; Stream and UDP. Stream connections are by far the most commonly used, but both types are supported by Dyalog APL.

Stream Sockets

A Stream socket is a connection-based transport that is analogous to a telephone service. A Stream socket handles error correction, guarantees delivery, and preserves data sequence. This means that if you send two messages to a recipient, the messages are sure to arrive and in the sequence that you sent them. However, individual messages may be broken up into several packets (or accumulated into one), and there is no predetermined protocol to identify message boundaries. This means that Stream-based applications must implement some kind of message protocol that both ends of a connection understand and adhere to.

User Datagram Protocol (UDP)

User Datagram Protocol (UDP) is a connection-less transport mechanism that is somewhat similar to a postal service. It permits a sending application to transmit a message or messages to a recipient. It neither guarantees delivery nor acknowledgement, nor does it preserve the sequence of messages. Messages are also limited to fit into a single packet which is typically no more than 1500 bytes in size. However, a UDP message will be delivered in its entirety.

You may wonder why anybody would use a service that does not guarantee delivery. The answer is that although UDP is technically an unreliable service, it is perfectly possible to implement reliable applications on top of it by building in acknowledgements, time-outs and re-transmissions.

Clients and Servers

A Stream based TCP/IP connection has two endpoints one of which is called the *server* and the other the *client*. However, this distinction is only relevant in describing how the connection is made.

The server initiates a connection by creating a socket which is identified by its (local) IP address and port number. The server is effectively making its service available to any client that wishes to connect. Notice that the server does not, at this stage, specify in any way which client or clients it will accept.

A client connects to a server by creating its own socket, specifying the IP address and port number of the service to which it wishes to connect.

Once the connection is established, both ends are capable of sending and receiving data and the original client/server relationship need no longer apply. Nevertheless, certain protocols, such as HTTP, do maintain the client/server model for the duration of the connection.

APL as a TCP/IP Server

A Stream based APL server initiates a connection by creating a TCPSocket object whose SocketType is `'Stream'` (the default).

The service is uniquely identified on the network by the server's IP Address and port number which are specified by the LocalAddr and LocalPort properties respectively. Note that unless you have more than one network adapter in your computer, LocalAddr is normally allowed to default.

This TCPSocket object effectively defines the availability of a particular service and at this stage is known as a *listening socket* which is simply waiting for a client to connect. This is reflected by the value of its CurrentState property which is `'Listening'`.

For example:

```
      'S0'⎕WC'TCPSocket' ('LocalPort' 2001)
      'S0'⎕WG'CurrentState'
 Listening
```

When a client connects to the APL server, the state of the TCPSocket object (which is reported by the CurrentState property) changes from `'Listening'` to `'Connected'` and it generates a TCPAccept event. Note that the connection cannot be nullified by the return value of a callback function attached to this event.

At this point, you can identify the client by the value of the RemoteAddr property of the TCPSocket object. If you wish to reject a particular client, you must immediately expunge the (connected) TCPSocket and then create a new one ready for another client.

Serving Multiple Clients

Dyalog APL provides a special mechanism to enable a single server to connect to multiple clients. This mechanism is designed to accommodate the underlying operation of the Windows socket interface in the most convenient manner for the APL programmer.

What actually happens when a client connects to your server, is that Windows automatically creates a new socket, leaving the original server socket intact, and still listening. At this stage, APL has a single name (the name of your TCPSocket object) but two sockets to deal with.

As it would be inappropriate for APL itself to assign a new name to the new socket, it disassociates the TCPSocket object from its original socket handle, and re-associates it with the new socket handle. This is reflected by a corresponding change in its SocketNumber property. The original listening socket is left, temporarily, in a state where it is not associated with the name of any APL object.

Having performed these operations, APL checks to see if you have attached a callback function to the TCPAccept event. If not, APL simply closes the original listening socket. This then satisfies the simple case where the server is intended to connect with only a single client and your socket has simply changed its state from `'Listening'` to `'Connected'`.

If there *is* a callback function attached to the TCPAccept event, APL invokes it and passes it the window handle of the listening socket. What the callback must do is to create a new TCPSocket object associated with this handle. If the callback exits without doing this, APL closes the original listening socket thereby preventing further clients from connecting.

If you wish to serve multiple clients, you must continually allocate new TCPSocket objects to the listening socket in this way so that there is always one available for connection.

The following example illustrates how this is done. Note that when the callback creates the new TCPSocket object, you **must not** specify any other property except SocketNumber, Event and Data in the `⎕WC` statement that you use to create it. This is important as the objective is to associate your new TCPSocket object with the original listening socket whose IP address and port number must remain unaltered.

Example

The original listening socket is created with the name `S0` and with a callback function `ACCEPT` attached to the TCPAccept event. The `COUNT` variable is initialised to `0`. This variable will be incremented and used to generate new names for new TCPSocket objects as each client connects.

```
      COUNT←0
      'S0'⎕WC'TCPSocket' ('LocalPort' 2001)
                         ('Event' 'TCPAccept' 'ACCEPT')
```

Then, each time a client connects, the `ACCEPT` function clones the original listening socket with a sequence of new TCPSocket objects using the name `S1`, `S2`, and so forth.

```
      ∇ ACCEPT MSG
[1]    COUNT+←1
[2]    ('S',⍕COUNT)⎕WC 'TCPSocket'('SocketNumber'(3⊃MSG))
      ∇
```

APL as a TCP/IP Client

A Stream based APL client makes contact with a server by creating a TCPSocket object whose SocketType is `'Stream'` (the default), specifying the RemoteAddr and RemotePort properties which identify the server's IP Address and port number respectively. Note that the client must know the identity of the server in advance.

For example:

```
IP←'193.32.236.43'
'CO'⎕WC'TCPSocket'('RemoteAddr 'IP)
                  ('RemotePort' 2001)
```

If the values of the RemoteAddr and RemotePort properties match the IP address and port number of any listening socket on the network, the association is made and the client and server sockets are connected.

When the connection succeeds, the state of the client TCPSocket object (which is reported by the CurrentState property) changes from `'Open'` to `'Connected'` and it generates a TCPConnect event. Note that the connection cannot be nullified by the return value of a callback function.

Host and Service Names

Although basic TCP/IP sockets must be identified by IP addresses and port numbers, these things are more commonly referred to by host and service names.

For example, the AltaVista web search engine is more easily identified and remembered by its name *www.altavista.com* than by any one of its IP addresses.

Port numbers are also often referred to by service names which are more convenient to remember. Furthermore, port numbers, even the so-called *well-known port numbers*, sometimes change, and your application will be more robust and flexible if you use names rather than hard-coded port numbers.

The WinSock API provides functions to resolve host names to IP addresses and service names to port numbers and these facilities are included in the Dyalog APL TCP/IP support.

Name resolution, in particular the resolution of host names, is performed *asynchronously*. This means that an application requests that a name be resolved, and then receives a message some time later reporting the answer. The asynchronous nature of name resolution is reflected in the way it is handled by Dyalog APL. Note that in certain cases, the resolution of a host name may take several seconds.

Each of the properties LocalPort, RemotePort, LocalAddr and RemoteAddr has a corresponding *name* property, i.e. LocalPortName, RemotePortName, LocalAddrName and RemoteAddrName. When you create a TCPSocket object, you may specify one or the other, but not both. For example, wherever you would use RemoteAddr, you may use RemoteAddrName instead.

If you use a *name* property, when you create a TCPSocket object, the TCPSocket will raise a TCPGotAddr or TCPGotPort event when the name is resolved to an IP address or a port number respectively. There is no need to take any action when these events are raised, so there is no specific need to attach callback functions. However, it may be useful to do so in order to track the progress of the requested connection.

The use of RemoteAddrName and TCPGotAddr is illustrated by the `BROWSER.QUERY` function that is described in the next Chapter.

Sending and Receiving Data

Once your TCPSocket object is connected, you can send and receive data. It makes no difference whether it was originally a server or a client; the mechanisms for data transfer are the same.

The *type* of data that you can send and receive is defined by the Style property which was established when you created the TCPSocket object. The default Style is `'Char'` which allows you to send and receive character vectors. Conversion to and from your ⎕AV is performed automatically.

If you choose to set Style to `'Raw'`, you can send and receive data as integer vectors whose values are in the range -127 to 255. This allows you to avoid any character translation.

If you set Style to `'APL'`, you may transmit and receive arbitrary arrays, including arrays that contain ⎕OR's of namespaces. Furthermore, however the data is actually fragmented by TCP/IP, an array transmitted in this way will appear to be sent and received in single atomic operation. Data buffering is handled automatically by APL itself. Style `'APL'` is normally only appropriate for communicating between two Dyalog APL sessions. Note however, that there is no mechanism to ensure that both ends of the connection use the same Style.

To send data, you execute the TCPSend method. For example, the following expression will transmit the string "Hello World" to the remote task connected to the TCPSocket object S0:

```
    2 ⎕NQ'S0' 'TCPSend' 'Hello World'
```

To receive data, you must attach a callback function to the TCPRecv event. Note that for a Stream connection you are not guaranteed to receive a complete message as transmitted by the sender. Instead, the original message may be received as separate packets or several messages may be received as a single packet. This means that you must perform your own buffering and you must implement a specific protocol to recognise message boundaries.

Output Buffering

When you use TCPSend to transmit a block of data, APL copies the data into a buffer that it allocates *outside* your workspace from Windows memory. APL then asks TCP/IP to send it.

However, the amount of data that can be transmitted in one go is limited by the size of various TCP/IP buffers and the speed of the network. Unless the block is very small, the data must be split up and transmitted bit by bit in pieces. This process, which is handled by APL in the background, continues until the entire data block has been transmitted. It could be several seconds or even minutes after you execute TCPSend before the entire block of data has been sent from your PC.

If in the meantime you call TCPSend again, APL will allocate a second buffer in Windows memory and will only try to send the second block of data when all of the first block has been transmitted.

If you call TCPSend repeatedly, APL will allocate as many buffers as are required. However, if you attempt to send too much data too quickly, this mechanism will fail if there is insufficient Windows memory or disk space to hold them.

If you need to transmit a very large amount of data, you should break it up into chunks and send them one by one. Having sent the first chunk, you can tell when the system is ready for the next one using the TCPReady event. This event is reported when the TCP/IP buffers are free *and* when there is no data waiting to be sent in the internal APL buffers. You should therefore attach a callback, whose job is to send the next chunk of data, to this event.

Note that a further level of buffering occurs in the *client* if the Style property of the TCPSocket is set to `'APL'`. This is done to prevent the partial reception of an APL array which would represent an invalid data object.

User Datagram Protocol (UDP) and APL

You may communicate with another application with User Datagram Protocol (UDP) by creating a TCPSocket object whose SocketType is `'UDP'`. For two APL applications to exchange data in this way, each one must have a UDP TCPSocket.

You make a UDP socket by creating a TCPSocket object, specifying the LocalAddr and LocalPort properties, and setting the SocketType to `'UDP'`. Unless your computer has more than one network card (and therefore more than one IP address), it is sufficient to allow LocalAddr to assume its default value, so in practice, only the port number is required. For example:

```
'S0' ⎕WC 'TCPSocket' ('LocalAddr' '') 2001
        ('SocketType' 'UDP')
'S0'⎕WG'CurrentState'
Bound
```

Once you have created a UDP TCPSocket, it is ready to send and receive data.

To send data to a recipient, you use the TCPSend method, specifying its RemoteAddr and RemotePort. The data will only be received if the recipient has a UDP socket open with the corresponding IP address and port number. However, note that there is no *absolute guarantee* that the recipient will ever get the message, nor, if you send several messages, that they will arrive in the order you sent them.

For example, the following statement might be used to send the character string `'Hello'` to a UDP recipient whose IP address is 123.456.789.1 and whose port number is 2002:

```
2 ⎕NQ 'S0' 'TCPSend' 'Hello' '123.456.789.1' 2002
```

Note that the maximum length of a UDP data packet depends upon the type of your computer, but is typically about 1500 bytes.

To receive data from a UDP sender, you must attach a callback to the TCPRecv event. Then, when the data is received, your callback function will be invoked. The event message passed as the argument to your callback will contain not only the data, but also the IP address and port number of the sender.

For example, if you created a TCPSocket called S1 as follows:

```
'S1' ⎕WC 'TCPSocket' ('LocalAddr' '') 2002
       ('SocketType' 'UDP')
       ('Event' 'TCPRecv' 'RECEIVE')
```

Where the callback function RECEIVE is as follows:

```
      ∇ RECEIVE MSG
[1]    DISPLAY MSG
      ∇
```

the following message would be displayed in your Session when the message 'Hello' was received from a sender whose IP address is 193.32.236.43 and whose port number is 2001.

```
.→------------------------------------------------.
| .→-. .→------. .→----. .→------------.          |
| |S1| |TCPRecv| |Hello| |193.32.236.43| 2001     |
| '--' '-------' '-----' '-------------'          |
'∊------------------------------------------------'
```

Client/Server Operation

We have seen how Dyalog APL may act as a TCP/IP server and as a TCP/IP client. It follows that full client/server operation is possible whereby an APL client workspace can execute code in an APL server workspace on the same or on a different computer.

A deliberately simple example of client/server operation is provided by the workspace `samples\tcpip\rexec.dws` whose operation is described below.

A more complex example, which implements a client/server APL component file system, is illustrated by the `samples\tcpip\qfiles.dws` workspace. See `DESCRIBE` in this workspace for details.

REXEC contains a small namespace called `SERVER`.

To start a server session, start Dyalog APL, and type:

```
)LOAD REXEC
SERVER.RUN
```

To use the server from an APL client, start Dyalog APL (on the same computer or on a different computer), and type:

```
)LOAD REXEC
IP SERVER.EXECUTE expr
```

where `IP` is the IP Address of the server computer and `expr` is a character vector containing an expression to be executed.

If you are testing this workspace using two APL sessions on the same computer, you can use either `'127.0.0.1'` or the result of the expression (`2 ⎕NQ '.' 'TCPGetHostID'`) for `IP`. This expression simply obtains the IP Address of your computer. Note however, that you **do** have to have an IP Address for this to work.

The RUN function

```
      ∇ RUN;CALLBACKS
[1]    ⎕EX↑'TCPSocket'⎕WN''
[2]    CALLBACKS←⊂('Event' 'TCPAccept' 'ACCEPT')
[3]    CALLBACKS,←⊂('Event' 'TCPRecv' 'RECEIVE')
[4]    COUNT←0
[5]    'S0'⎕WC'TCPSocket' ''PORT('Style' 'APL'),CALLBACKS
      ∇
```

RUN[1]   expunges all TCPSocket objects that may be already defined. This is
            intended only to clear up after a potential error.

RUN[2-3]   set up a variable CALLBACKS which associates various functions
            with various events.

RUN[4]   initialises a variable COUNT which will be incremented and used to
            name new TCPSocket objects as each client connects. COUNT is
            global within the SERVER namespace.

RUN[5]   creates the first TCPSocket server using your default IP address and
            the port number specified by the PORT variable (5001). Note that the
            Style property is set to 'APL' so that data is transmitted and
            received in internal APL format. Furthermore, however each message
            gets fragmented by TCP/IP, it will always appear to be sent and
            received in an atomic operation. There is no need for the client to do
            any buffering.

Once the server has been initiated, the next stage of the process is that a client makes a
connection. This is handled by the ACCEPT callback function.

The ACCEPT function

```
      ∇ ACCEPT MSG;SOCK;EV
[1]    COUNT←COUNT+1
[2]    SOCK←'SocketNumber'(3⊃MSG)
[3]    EV←'Event'((⊃MSG)⎕WG'Event')
[4]    ('S',⍕COUNT)⎕WC'TCPSocket'SOCK EV
      ∇
```

The ACCEPT function is invoked when the TCPAccept event occurs. This happens when a client connects to the server.

Its argument MSG, supplied by APL, is a 3-element vector containing:

MSG[1] The name of the TCPSocket object
MSG[2] The name of the event ('TCPAccept')
MSG[3] The socket handle for the original listening socket

ACCEPT[1]   increments the COUNT variable. This variable is global to the
                SERVER namespace and was initialised by the RUN function.

ACCEPT[4]   makes a new TCPSocket object called Sxx, where xx is the new value
                of COUNT. By specifying the socket handle of the original listening
                socket as the value of the SocketNumber property for the new object,
                this effectively clones the listening socket. Note that the cloned socket
                inherits 'Style' 'APL'. For further discussion of this topic, see
                Serving Multiple Clients.

The RECEIVE function

```
      ∇ RECEIVE MSG;RSLT
[1]     :Trap 0
[2]         RSLT←0('#'⍎(3⊃MSG))
[3]     :Else
[4]         RSLT←⎕EN
[5]     :EndTrap
[6]     2 ⎕NQ(⊃MSG)'TCPSend'RSLT
      ∇
```

The `RECEIVE` function is invoked when the TCPRecv event occurs. This happens whenever an APL array is received from a client. Note that it is guaranteed to receive an entire APL array in one shot because the Style property of the TCPSocket object is set to `'APL'`.

Its argument `MSG` , supplied by APL, is a 5-element vector containing:

`MSG[1]` The name of the `TCPSocket` object
`MSG[2]` The name of the event ('TCPRecv')
`MSG[3]` The data
`MSG[4]` IP address of the client
`MSG[5]` Port number of the client

`RECEIVE[1-5]` executes the expression `(3⊃MSG)` received from the client. Assuming it succeeds, `RSLT` is a 2-element vector containing a zero followed by the result of the expression. If the execute operation fails for any reason, `RSLT` is set to the value of `⎕EN` (the error number).

`RECEIVE[6]`   transmits the result back to the client.

The EXECUTE function

```
      ∇ RSLT←SERVER_IP EXECUTE EXPR;P;SOCK
[1]    ⍝ Execute expression in server
[2]
[3]    P←⊂'TCPSocket'
[4]    P,←⊂'RemoteAddr'SERVER_IP  ⍝ IP Address
[5]    P,←⊂'RemotePort'PORT       ⍝ Port Number
[6]    P,←⊂'Style' 'APL'
[7]    P,←⊂'Event'('TCPRecv' 1)('TCPClose' 1)('TCPError' 1)
[8]    'SOCK'⎕WC P
[9]
[10]   2 ⎕NQ'SOCK' 'TCPSend'EXPR ◇ RSLT←⎕DQ'SOCK'
[11]
[12]   :Select 2⊃RSLT
[13]   :Case 'TCPRecv'
[14]       RSLT←3⊃RSLT
[15]       :If 0=⊃RSLT
[16]           RSLT←2⊃RSLT
[17]       :Else
[18]           ('Server: ',⎕EM RSLT)⎕SIGNAL RSLT
[19]       :EndIf
[20]   :Case 'TCPError'
[21]       ('Server Error: ',,⎕FMT 2↓RSLT)⎕SIGNAL 201
[22]   :Else
[23]       'Unknown Server Error'⎕SIGNAL 201
[24]   :EndSelect
      ∇
```

This function is executed by a client APL session. Its right argument is a character vector containing an expression to be executed. Its left argument is the IP Address of a server APL session in which the expression is to be run. The server session may be running on the same computer or on a different computer on the network.

EXECUTE[3-8] makes a client TCPSocket object called SOCK for connection to the specified server IP address and port number PORT. Note that the Style property is set to 'APL' so that data is transmitted and received in internal APL format. Furthermore, however each message gets fragmented by TCP/IP, it will always appear to be sent and received in an atomic operation. There is no need for the client to do any buffering.

The Event property is set so that events TCPRecv, TCPClose and TCPError will terminate the ⎕DQ. In this case, this is easier than using callback functions.

`EXECUTE[10]` transmits the expression to the server for execution and then `⎕DQ`s the socket. As the only events enabled on the socket are TCPRecv, TCPClose and TCPError it effectively *waits* for one of these to occur. When one of these events does happen, the `⎕DQ` terminates, returning the corresponding event message as its result.

The reason for using a diamond expression is to ensure that the TCPRecv, TCPClose or TCPError event will not be fired before the `⎕DQ` was called.

A second point worth noting is that the TCPSend request is automatically queued until the socket gets connected. In this case, there is no need to trigger the TCPSend from a callback on the TCPConnect event.

`EXECUTE[12-]` process the TCPRecv, TCPClose or TCPError event that was generated by the socket. If the operation was successful, `RSLT[2]` contains `'TCPRecv'` and `RSLT[3]` contains a zero followed by the result of the expression.

CHAPTER 10

APL and the Internet

Introduction

This chapter describes the use of TCPSocket objects to access the internet. From Version 12.0, a new tool called Conga is the recommended mechanism for connecting to the internet. The code samples in this chapter have not been ported to the Unicode Edition, but continue to work in Classic Editions. For information on accessing the internet and other TCP services in the Unicode Edition, see the *Conga User Guide*.

A complete description of how Web browsers and servers work is beyond the scope of this document. Nevertheless, the following basic introduction should prove a useful introduction before trying to write a server or client application in Dyalog APL.

A Web server is simply a TCP/IP server that adheres to a particular protocol known as Hypertext Transfer Protocol (HTTP). Every request for a document from a Web browser to a Web server is a new connection. When a Web browser requests an HTML document from a Web server, the connection is opened, the document transferred, and the connection closed.

The Web server advertises its availability by opening a Stream socket. Conventionally, it uses Port Number 80 although other port numbers may be used if and when required.

A client (normally referred to as a *browser*) connects to a server and immediately sends it a *command*. A command is a text string containing sub-strings separated by CR,LF pairs and terminated by CR,LF (an empty line). This terminator is an **essential** part of the protocol as it notifies the server that the entire command has been received. An example of a command sent by Netscape Navigator 3.0 Gold is:

```
GET / HTTP/1.0<CR,LF>
Proxy-Connection: Keep-Alive<CR,LF>
User-Agent: Mozilla/3.0Gold (Win95; I) <CR,LF>
Host: pete.dyalog.com<CR,LF>
Accept: image/gif, image/x-xbitmap, image/jpeg,
                 image/pjpeg, */*<CR,LF>
<CR,LF>
```

The first line of the command is a statement that tells the server what the client wants. The simplest statement is of the form GET <url>, which instructs the server to retrieve a particular document. In the example, <url> is "/" which is a relative Universal Resource Locator (URL) that identifies the home page of the current server. The client may also specify the level of http protocol that it understands as a second parameter. In the above example, the client is requesting HTTP version 1.0. Subsequent statements provide other information that may be useful to the server.

The server receives the command, actions it, and then sends back the result; in this case, the content of the Web page associated with the given URL. Using the original HTTP version 1.0 protocol, the server then **closes** the TCP/IP socket. This act informs the client that all of the data has been received and that the entire transaction is complete.

Today's web servers commonly use HTTP 1.1 which supports persistent connections. This means that the socket may not be closed, but is instead left open (for a time) for potential re-use. This behaviour is specified by *Connection: Keep-Alive HTTP headers* which are beyond the scope of this discussion. However, to support persistent connections, even the simplest client must be able to detect that the transaction is complete in some other way. A simple solution, as implemented in the BROWSER.QUERY function, is to look for the HTML end-tag.

The protocol can therefore be summarised as:

        a) Client connects to server
        b) Client sends command (terminated by CR,LF)
        c) Server sends requested data to client
        d) Server disconnects from client (HTTP 1.0 only)

A Web page normally contains text and embedded hyperlinks which connect it to other WWW pages. When the user activates a hyperlink, the browser connects to the corresponding server and requests the relative URL.

However, if you are using a secure proxy server, as most corporate users do, the browser connects repeatedly to your proxy (rather than to specific servers) and requests the *absolute* URL (which contains the name of the server) instead.

Writing a Web Client

A sample Web client is provided in the `BROWSER` namespace in the workspace `samples\tcpip\www.dws`.

Before you can use `BROWSER.QUERY` you must be connected to the Internet. See *APL and the Internet* for details.

The main function is `BROWSER.QUERY`. This function is intended to be used in one of two ways:

Using a Proxy Server

If you are connected to the Internet through a secure proxy server or *firewall* (as is common in many commercial organisations), you may **only** connect to your firewall; you cannot connect directly to any other server. Effectively, the **only external** IP address to which you may connect a TCPSocket as a client is the IP address of your firewall.

In this case, you should set the values of the variables `BROWSER.IP_ADDRESS` and `BROWSER.PORT_NUMBER` to the IP address and port number of your firewall.

The right argument to `BROWSER.QUERY` is a character string that includes the name of the web site or server as part of the query. For example, the following statement will retrieve the Microsoft home page:

```
BROWSER.QUERY'GET http://www.microsoft.com/'
```

Using a Direct Connection

If you are directly connected to the Internet or you use dial-up networking to connect to an Internet provider, you may create TCPSocket objects that are directly connected to any server on the Internet.

In this case, the *left* argument to the function is the address and port number of the server to which you wish to connect (the port number is optional and defaults to 80). The *right* argument is the command that you wish the server to execute. Furthermore, the address may be expressed as the *IP address* of the server or as the *name* of the server.

For example, to obtain the Microsoft home page :

```
        '207.46.192.254'BROWSER.QUERY'GET /'
or
        'www.microsoft.com'BROWSER.QUERY'GET /'
```

The result of the query is not returned by the `BROWSER.QUERY` function, but is instead obtained from the server *asynchronously* by callback functions and then deposited in the variable `BROWSER.HTML`. In this example, the call-backs report the progress of the transaction as shown below. This approach is perhaps unusual in APL, but it perfectly illustrates the event-driven nature of the process.

Using a Firewall

```
      BROWSER.QUERY 'GET http://www.microsoft.com'
Connected to 193.32.236.22
... Done
Received 39726 Bytes
Response is in:
#.BROWSER.HTML
```

Using a Direct Connection

```
      'www.microsoft.com' BROWSER.QUERY 'GET /'
www.microsoft.com resolved to IP Address 207.46.192.254
Connected to 207.46.192.254
... Done
Received 39726 Bytes
Response is in:
#.BROWSER.HTML
```

There are two points to note. In the first case (using a firewall) the IP address reported is the IP address of your firewall. In the second case, there is an additional first step involved as the name of the server is resolved to its IP address (note too that this web site provides a number of IP addresses).

To keep the examples simple, `BROWSER.QUERY` has been written to handle only a single query at a time. Strictly speaking, it could initiate a second or third query before the result of the first had been received. This would merely entail creating multiple sockets instead of a single one.

The various functions in the `BROWSER` namespace are as follows:

QUERY	User function to initiate a Web query
GOTADDR	callback: reports name resolution (server name to IP address)
CONNECT	callback: handles the connection to the server
RECEIVE	callback: collects the data packets as they arrive from the server
CLOSE	callback: stores the result of the query and expunges TCPSocket
ERROR	callback: handles errors

The QUERY function

```
      ∇ {LARG}QUERY QRY;IP;PN;CALLBACKS;NS;P;SERVER
[1]    ⍝ Perform world wide web query
[2]     :If 0=⎕NC'LARG'
[3]         IP←IP_ADDRESS
[4]         PN←PORT_NUMBER
[5]         QRY,←' HTTP/1.0',⎕AV[4 3 4 3]
[6]     :Else
[7]         :If (¯2≡≡LARG)^(,2)≡⍴LARG
[8]             IP PN←LARG
[9]         :Else
[10]            IP PN←LARG 80
[11]        :EndIf
[12]        QRY,←' HTTP/1.1',⎕AV[4 3],'Host:',IP,4⍴⎕AV[4 3]
[13]    :EndIf
[14]
[15]   ⍝ Server specified by name or IP address ?
[16]    :If ^/IP∊'. ',⎕D
[17]        SERVER←('RemoteAddr'IP)
[18]    :Else
[19]        SERVER←('RemoteAddrName'IP)
[20]    :EndIf
[21]
[22]    NS←(''⎕NS''),'.'
[23]    CALLBACKS←⊂('TCPGotAddr'(NS,'GOTADDR'))
[24]    CALLBACKS,←⊂('TCPConnect'(NS,'CONNECT'))
[25]    CALLBACKS,←⊂('TCPRecv'(NS,'RECEIVE'))
[26]    CALLBACKS,←⊂('TCPClose'(NS,'CLOSE'))
[27]    CALLBACKS,←⊂('TCPError'(NS,'ERROR'))
[28]
[29]   ⍝ Expunge TCPSocket in case of previous error
[30]    ⎕EX'SO'
[31]
[32]   ⍝ Make new SO namespace containing QRY
[33]    'SO'⎕NS'QRY'
[34]   ⍝ Then make SO a TCPSocket object
[35]    P←SERVER('RemotePort'PN)('Event'CALLBACKS)
[36]    SO.⎕WC(⊂'TCPSocket'),P ◇ ⎕DQ'SO'
      ∇
```

The first 13 lines of the function process the optional left argument and are largely unremarkable.

However, note that if you are using a firewall or proxy (no left argument), `QUERY[5]` adds a header to request HTTP/1.0 protocol. If you are using a direct connection, `QUERY[12]` instead adds a request for HTTP/1.1 protocol and a *Host* header (which in this case it knows). A Host header is mandatory for HTTP/1.1 and your firewall may add one for you.

QUERY[16-20] sets the variable SERVER to specify either RemoteAddr or RemoteAddrName according to whether the user specified the IP address or the name of the server.

QUERY[22-27] set up a variable CALLBACKS which associates various functions with various events. Full path-names are used for the callback functions because they will be associated by a ⎕WC statement that is executed *within* the S0 namespace.

QUERY[30] expunges the object S0. This is done only in case an error occurred previously and the object has been left around.

QUERY[33] makes a new namespace called S0 and copies the variable QRY into it. This is done because the query cannot be submitted to the server until after a connection has been made. Thus the query is encapsulated within the TCPSocket object to make it available to the callback function CONNECT that will handle this event. A less elegant solution would be to use a global variable.

QUERY[36] creates a new client TCPSocket object associated with the namespace S0.

A more obvious solution would be..

```
[33]    'S0' ⎕WC(⊂'TCPSocket'),P
[34]    S0.QRY←QRY
```

However, this is inadvisable because TCP events can occur as soon as the object has been created. If the TCPConnect event fired before QUERY[34] could be executed, the CONNECT callback function would fail with a **VALUE ERROR** because S0.QRY would not yet be defined. This is also a reason for attaching the callback functions in the ⎕WC statement and not in a subsequent ⎕WS. You do not want the TCPConnect event to fire when there is no callback attached.

Note that these timing issues are only relevant because **BROWSER.QUERY** is a user-called function and not a callback. If it were a callback, APL would automatically queue the incoming TCP events until it (the callback) had terminated.

Depending upon how the function was called, the next part of the process is handled by the GOTADDR or the CONNECT callback.

The GOTADDR callback function

```
     ∇ GOTADDR MSG;NAME;IP
[1]    NAME IP←(⊃MSG)⎕WG'RemoteAddrName' 'RemoteAddr'
[2]    NAME,' resolved to IP Address ',IP
     ∇
```

The GOTADDR callback function is invoked when the TCPGotAddr event occurs. This happens if the RemoteAddrName was specified when the TCPSocket was created.

The function merely obtains the name and newly resolved IP address of the server from the RemoteAddrName and RemoteAddr properties of the TCPSocket object and reports them in the session.

The CONNECT callback function

```
      ∇ CONNECT MSG
[1]    ⎕CS⊃MSG
[2]    'Connected to ',⎕WG'RemoteAddr'
[3]    BUFFER←''
[4]    2 ⎕NQ'' 'TCPSend' QRY
      ∇
```

The CONNECT function is invoked when the TCPConnect event occurs. This happens when the server accepts the client.

Its argument MSG , supplied by APL, is a 2-element vector containing:

MSG[1] The name of the TCPSocket object
MSG[2] The name of the event ('TCPConnect')

CONNECT[1]   changes to the namespace of the TCPSocket object.

CONNECT[2]   displays the IP address of the server to which the client has successfully connected. This is obtained from the RemoteAddr property of the TCPSocket object.

CONNECT[3]   initialises a variable BUFFER which will be used to collect incoming data from the server. Notice that as the function has changed to the TCPSocket namespace, this variable is encapsulated within it rather than being global.

CONNECT[3]   uses the TCPSend method to send the query (the QRY variable was encapsulated within the TCPSocket object when it was created by the QUERY function) to the server.

The next part of the process is handled by the RECEIVE callback.

The RECEIVE callback function

```
      ∇ RECEIVE M
[1]    ⎕CS⊃M
[2]    BUFFER,←3⊃M
[3]    :If ∨/'</html>'∊##.lcase 3⊃M
[4]        (⊃M)##.⎕WS'TargetState' 'Closed'
[5]    :EndIf
      ∇
```

The `RECEIVE` function is invoked when the TCPRecv event occurs. This happens whenever a packet of data is received. As the response from the server can be of arbitrary length, the job of the `RECEIVE` function is simply to collect each packet of data into the `BUFFER` variable.

Its argument `MSG` , supplied by APL, is a 3-element vector containing:

`MSG[1]` The name of the `TCPSocket` object
`MSG[2]` The name of the event ('TCPRecv')
`MSG[3]` The data
`MSG[4]` IP address of the client
`MSG[5]` Port number of the client

`RECEIVE[1]`   changes to the namespace of the TCPSocket object.

`RECEIVE[2]`   catenates the data onto the variable `BUFFER`, which is encapsulated within the TCPSocket object.

`RECEIVE[3]`   checks for the presence of an *end-of document* HTML tag, which indicates that the entire page has arrived, and if so

`RECEIVE[4]`   sets the TargetState property of the TCPSocket to `'Closed'`. This initiates the closure of the socket. Although a client can close a socket by expunging the associated TCPSocket namespace, our data (`BUFFER`) is in this namespace and we do not want to lose it.

Note that it is necessary for `RECEIVE` to detect the end-of-document in this way, so as to support HTTP/1.1 protocol.

The CLOSE callback function

```
     ∇ CLOSE MSG
[1]    HTML←(⊃MSG)⍎'BUFFER'
[2]    ⎕EX⊃MSG
[3]    '... Done'
[4]    'Received ',(⍕⍴HTML),' Bytes'
[5]    'Response is in:'
[6]    (''⎕NS''),'.HTML'
     ∇
```

The `CLOSE` function is invoked when the TCPClose event occurs. This happens when the server closes the socket. If the protocol is HTTP/1.0, this will be done immediately after the server has sent the data in response to the query. If the protocol is HTTP/1.1, the closure may be performed at the request of the client.

Note that the data has been buffered by the `RECEIVE` function as it arrived.

Its argument `MSG` , supplied by APL, is a 2-element vector containing:

`MSG[1]`	The name of the `TCPSocket` object
`MSG[2]`	The name of the event (`'TCPClose'`)

`CLOSE[1]`	copies the contents of the `BUFFER` variable (that is local to the TCPSocket object) into the `HTML` variable that is global within the current (`BROWSER`) namespace. (Clearly this design would be inadequate if `BROWSER` was extended to support multiple concurrent queries.)
`CLOSE[2]`	expunges the TCPSocket object
`CLOSE[3-6]`	reports a successful end to the query and displays the size of the result.

The ERROR callback function

```
      ∇ R←ERROR MSG
[1]    DISPLAY MSG
[2]    ⎕EX⊃MSG
[3]    R←0
      ∇
```

The ERROR function is invoked if and when a TCP/IP error occurs.

Its argument MSG , supplied by APL, is a 4-element vector containing:

MSG[1]   The name of the TCPSocket object
MSG[2]   The name of the event ('TCPError')
MSG[3]   An error number
MSG[4]   An error message

ERROR[1]   displays the contents of MSG using the DISPLAY function.

ERROR[2]   expunges the TCPSocket object (it is no longer usable)

ERROR[3]   returns a 0. This tells APL *not* to perform the normal default
                processing for this event, which is to display a message box.

Writing a Web Server

A sample Web server is provided in the `SERVER` namespace in the workspace `samples\tcpip\www.dws`. This is a deliberately over-simplified example to illustrate the principles involved. It is capable of handling concurrent connections from several clients, but (for simplicity) does not use multi-threading. A more comprehensive web server workspace, `aplserve\server.dws`, is also provided. This workspace is intended to be the basis of a production web server, and does use multithreading. Documentation is provided in the workspace itself.

The main function is `SERVER.RUN` which is niladic and initialises the APL Web server using your default IP Address and port number 81.

To use the server, you must start a Web browser such as Firefox or Microsoft Internet Explorer. You may do this on another PC on your network or on your own PC. If so, you will probably find it most convenient to position your Dyalog APL Session Window and your browser window alongside one another.

To connect to the server, simply enter your user name (or your IP address, or "127.0.0.1" or "localhost") followed by 81 in the appropriate field in your browser, and then press Enter. For example:

>   http://localhost:81

When you press Enter, your browser will try to connect with a server whose IP address is your IP address and whose port number is 81; in short, the APL server. Upon connection, the following messages (but with different IP addresses) will appear in your Session window:

```
        SERVER.RUN
Connected to 193.32.236.22
URL Requested:
Connected to 193.32.236.22
URL Requested: images/dyalog.gif
```

and the Dyalog APL home page will appear in your browser. This has in fact been supplied by your APL server.

The functions in the `SERVER` namespace are as follows:

`RUN`	user function to initiate an APL Web server
`ACCEPT`	callback which handles client connections
`RECEIVE`	callback which handles client commands
`ERROR`	callback which handles errors

The RUN function

```
      ∇ RUN;CALLBACKS
[1]    ⎕EX↑'TCPSocket'⎕WN''
[2]    CALLBACKS←⊂('Event' 'TCPAccept' 'ACCEPT')
[3]    CALLBACKS,←⊂('Event' 'TCPRecv' 'RECEIVE')
[4]    CALLBACKS,←⊂('Event' 'TCPError' 'ERROR')
[5]    COUNT←0
[6]    'S0'⎕WC'TCPSocket' '' 81,CALLBACKS
      ∇
```

RUN[1]   expunges all TCPSocket objects that may be already defined. This is
                intended only to clear up after a potential error.

RUN[2-4]   set up a variable CALLBACKS which associates various functions
                with various events.

RUN[5]   initialises a variable COUNT which will be incremented and used to
                name new TCPSocket objects as each client connects. COUNT is
                global within the SERVER namespace.

RUN[6]   creates the first TCPSocket server using your default IP address and
                port number 81.

Once the server has been initiated, the next stage of the process is that a client makes a
connection. This is handled by the ACCEPT callback function.

The ACCEPT callback function

```
      ∇ ACCEPT MSG;EV
[1]    COUNT←COUNT+1
[2]    EV←'Event'((⊃MSG)⎕WG'Event')
[3]    ('S',⍕COUNT)⎕WC'TCPSocket'('SocketNumber'(3⊃MSG))EV
[4]    ⎕CS⊃MSG
[5]    'Connected to ',(⎕WG'RemoteAddr')
[6]    BUFFER←⎕AV[4 3]
      ∇
```

The `ACCEPT` function is invoked when the TCPAccept event occurs. This happens when a client connects to the server.

Its argument `MSG` , supplied by APL, is a 3-element vector containing:

MSG[1]	The name of the `TCPSocket` object
MSG[2]	The name of the event (`'TCPAccept'`)
MSG[3]	The socket handle for the original listening socket

ACCEPT[1]	increments the `COUNT` variable. This variable is global to the `SERVER` namespace and was initialised by the `RUN` function.
ACCEPT[3]	makes a new TCPSocket object called Sxx, where xx is the new value of `COUNT`. By specifying the socket handle of the original listening socket as the value of the SocketNumber property for the new object, this effectively clones the listening socket. For further discussion of this topic, see Serving Multiple Clients.
ACCEPT[4]	changes to the namespace of the TCPSocket object, that is now connected to a client.
ACCEPT[5]	displays the message `Connected to xxx.xxx.xxx.xxx`, the IP address of the client, which is obtained from the value of the RemoteAddr property.
ACCEPT[6]	initialises a variable `BUFFER` to `⎕AV[4 3]` (CR,LF). This variable is global to the `SERVER` namespace and is used by the `RECEIVE` callback function to accumulate the command that is transmitted by the client. This happens next.

The RECEIVE callback function

```
      ∇ RECEIVE MSG;CMD;OLD;URL;FILE;DATA
[1]    OLD←⎕CS⊃MSG
[2]    BUFFER,←3⊃MSG
[3]    :If ⎕AV[4 3 4 3]≢¯4↑BUFFER ⍝ Have we all?
[4]        :Return
[5]    :EndIf
[6]
[7]    CMD←2↓¨(⎕AV[4 3]⍷BUFFER)⊂BUFFER
[8]    CMD←⊃CMD ⍝ Ignore everything except client request
[9]    ⎕CS OLD
[10]   :If 'GET /'≡5↑CMD
[11]       URL←5↓CMD
[12]       URL←(¯1+URLⵏ' ')↑URL
[13]       ⎕←'URL Requested: ',URL
[14]       :If 0=⍴URL ◇ URL←'index.htm' ◇ :EndIf
[15]       URL←(-'.html'≡¯5↑URL)↓URL
[16]       FILE←(2 ⎕NQ'.' 'GetEnvironment') 'Dyalog'
[17]       FILE,←HOME,URL
[18]       DATA←GETFILE FILE
[19]       DATA,←(0<⍴DATA)/'File not found'
[20]       2 ⎕NQ(⊃MSG)'TCPSend'DATA
[21]   :EndIf
[22]
[23]   :If 9=⎕NC⊃MSG ◇ (⊃MSG)⎕WS'TargetState' 'Closed' ◇
:EndIf
[24]   :If 9=⎕NC⊃MSG ◇ ⎕DQ⊃MSG ◇ :EndIf
      ∇
```

The `RECEIVE` function is invoked whenever a TCPRecv event occurs. This happens when a data packet is received from a client

Its argument `MSG` , supplied by APL, is a 3-element vector containing:

MSG[1]	The name of the `TCPSocket` object
MSG[2]	The name of the event (`'TCPReceive'`)
MSG[3]	The data

RECEIVE[1]	changes to the namespace of the TCPSocket object. The name of the current namespace is stored in the local variable `OLD`.
RECEIVE[2]	catenates the newly received data packet to the `BUFFER` variable that is encapsulated in the TCPSocket object and was initialised by the `ACCEPT` function.
RECEIVE[3-5]	tests whether or not all of the command sent by the client has been received. This is true only if the last 4 characters of `BUFFER` are CR,LF,CR,LF. If there is more data to come, `RECEIVE` exits; otherwise it goes on to process the command.

`RECEIVE[7-8]`	splits the command into sub-strings and then discards all but the first one.
`RECEIVE[9]`	changes back into the `SERVER` namespace
`RECEIVE[10-11]`	parses the client request for a URL. For the sake of simplicity, the request is assumed to begin with the string `'GET /'`. If not, the request is ignored.
`RECEIVE[12]`	removes all trailing information that might be supplied by the browser after the URL.
`RECEIVE[14]`	checks for a request for an empty URL (which equates to the home page). If so, it substitutes `index.htm` which is the name of the file containing the home page.
`RECEIVE[15]`	drops the file extension of the URL if `.html` to `.htm` if required.
`RECEIVE[16]`	sets the value of local variable `FILE` to the name of the directory in which Dyalog APL is installed.
`RECEIVE[17]`	appends the path-name of the sub-directory `\help` and the name of the URL. `FILE` now contains the full path-name of the requested web page file.
`RECEIVE[18]`	uses the utility function `GETFILE` to read the contents of the file into the local variable `DATA`.
`RECEIVE[19]`	checks that the result of `GETFILE` was not empty and if so, appends an appropriate message. This would be the case if the file did not exist.
`RECEIVE[20]`	uses TCPSend to transmit the contents of the file to the browser.
`RECEIVE[23-24]`	closes the TCPSocket object. This is a fundamental part of the HTTP protocol because when the client socket subsequently gets closed, it knows that all of the data transmitted by the server has been received. Notice that the function does not simply expunge the socket which could result in loss of yet untransmitted data. Instead, it closes the socket by setting its TargetState property to `'Closed'`, and then (if necessary) waiting. Once all the buffered data has been transmitted, the socket will be closed and the TCPSocket object will disappear. This causes the `⎕DQ` to terminate.

For further information on WWW protocol, see the *Introduction* to this chapter.

CHAPTER 11

OLE Automation Client and OLE Controls

Introduction

OLE Automation allows you to drive one application from another and to mix code written in different programming languages. In practical terms, this means that you may write a subroutine to perform calculations in (say) C++ and use the subroutine directly in Visual Basic 4 or Excel. Equally, you could write the code in Visual Basic and call it from C++. Dyalog APL/W is a fully subscribed member of this code-sharing club.

OLE Automation is, however, much more than just a mechanism to facilitate cross-application macros because it deals not just with subroutine calls but with *objects*. An object is a combination of code and data that can be treated as a unit. Without getting too deeply into the terminology, an object defines a *class*; when you work with an object you create one or more *instances* of that class.

This chapter describes how Dyalog APL can drive other applications using OLE Automation. In these circumstances, Dyalog APL is acting as an OLE *client*.

There are two types of OLE object involved; OLE Servers and ActiveX controls. An ActiveX control can be instantiated as a GUI object within a Dyalog APL Form, whereas an OLE Server either has no GUI component, or is a separate object. Otherwise, the two are very similar.

You can obtain lists of the OLE Servers and ActiveX Controls installed on your computer from the OLEServers and OLEControls properties of the system object `'.'`. These lists are obtained from your Windows Registry and therefore contains only those OLE objects that are correctly installed. Each OLE Server and OLE Control is identified by its name and class identifier. Either may be used to access it.

Using an OLE Server

You can access an OLE Automation or COM Server using the OLEClient object. When you create an OLEClient, you specify the name of the Server as the ClassName property for the object.

For example:

```
XL←⎕NEW 'OleClient' (⊂'ClassName' 'Excel.Application')
```

or, using ⎕WC

```
'EX'⎕WC'OLEClient' 'Excel.Application'
```

The effect of both statements is to create an object `EX`, which is connected to an instance of the of the Excel.Application Class, an OLE Server. The OLE Server instance may be *in-process* or *out-of-process*. If it is in-process, the code and data associated with the instance are loaded into the same address space as the Dyalog APL process. In the latter case, the instance represents a separate Windows process on your computer or, on an entirely different computer in the network.

When APL connects to an out-of-process OLE Server in this way, you can specify whether you wish to connect to an existing (running) instance of the Server, or start a new copy of the Server. This is done using the InstanceMode property.

Loading an ActiveX Control

An ActiveX or OLE Control is in fact a type of Dynamic Link Library (DLL) which must be loaded before it can be used. This is done by creating an OCXClass object using ⎕WC or ⎕NEW.

For example, if you have an OLE Control named "Microsoft Office Chart 9.0 ", you can load it with the following statements (which are split here only to prevent text wrap)

```
NAME←' Microsoft Office Chart 9.0 '
MOC←⎕NEW 'OCXClass' (⊂'ClassName' NAME)
```

or, using ⎕WC

```
'MOC' ⎕WC 'OCXClass' NAME
```

The right argument is a character string containing the name or class identifier of the ActiveX Control. The left argument is an arbitrary name of your own choosing by which you will subsequently refer to the Control class.

Using an OLE Control

Having created an OCXClass object, you may *use* an OLE Control by creating an *instance* of it from its class. The instance must be created as the child of a Form. For example:

```
'F' ⎕WC 'Form'
'F.MM' ⎕WC 'MOC' ⍝ Instance of MOC
```

Although you can obtain general information about an OLE Control from both the class (represented by the OCXClass object) and any instance, you may only query and manipulate a control through an instance.

Type Information

In general, it is a requirement that all COM objects provide *Type Information*. This is commonly provided in a type library file (extension .TLB) or is included in the object's .EXE or .DLL file. Type Information includes the names of the methods, events and properties exported by the object, together with descriptions of the arguments to each method and descriptions of the data types of each property. Type Information is necessary for the COM object to be properly recognised by object browsers and by application development systems.

When you load a COM object, APL reads all of the Type Information associated with the top-level object into the workspace. In addition, it reads the Type Information for all other objects in the same object hierarchy, and the Type Information for any other COM objects that are used or referenced by it. This Type Information is retained in the workspace when you)SAVE it. When you reattach an OLEClient or OCXClass to the same object, there is no need for the Type Information to be re-read. Because the operation to read the Type Information may take several seconds, possibly minutes, this design optimises run-time performance. Note however, that the Type Information does occupy a considerable amount of workspace.

Dyalog APL uses the Type Information to expose the names, data types and arguments of all the methods, events and properties provided by the object, and those of all the other sub-objects in the object hierarchy. Dyalog APL also uses the Type Information to validate the arguments you supply to methods (both the number and the data types) and the values you assign to properties. For example, if a method is defined to take an argument VT_I4, Dyalog APL will issue a `DOMAIN ERROR` if you invoke the method with a character argument. Internally, Dyalog APL uses the Type Information to convert between APL arrays and OLE data types.

Warning: not all COM objects provide Type Information, or do so in non-standard ways. Perhaps the reason for this omission is that is that **Microsoft Visual Basic for Applications (VBA) does not itself require Type Information**. OLE data types are for the most part identical to VBA data types. Furthermore, VBA syntax does not distinguish between calling a function and referencing a variable. Therefore, all you need to drive a COM object from VBA is the documentation. Finally, as most other Microsoft products use VBA as their programming interface, authors of COM objects can satisfy most of their potential users without Type Information and so take the easy way out. Nevertheless, OLE Servers which fail to provide Type Information *can* be successfully used from Dyalog APL; for details, see the section entitled *OLE Objects without Type Information* later in this Chapter.

Identifying Properties, Methods and Events

You can obtain the names of all the properties, methods, and events exposed by a COM object by executing the system function ⎕NL, with the appropriate argument, inside the namespace that is associated with an instance of the object. Note that the result of ⎕NL is a vector of character vectors. If Type Information is unobtainable, the list of items reported by ⎕NL will be empty. See the section entitled *OLE Objects without Type Information* later in this Chapter.

For example:

```
      DB←⎕NEW'OleClient' (⊂'ClassName' 'DAO.DBEngine.36')

      DB.⎕NL ¯3 ⍝ Methods
 BeginTrans  CommitTrans  CompactDatabase  CreateDatabase
      CreateWorkspace  Idle  ISAMStats  OpenConnection
      OpenDatabase  RegisterDatabase  RepairDatabase  Ro
      llback  SetOption

      DB.⎕NL ¯2 ⍝ Properties
 AutoBrowse  ChildList  ClassID  ClassName  Data  Default
      Password  DefaultType  DefaultUser  Errors  Event
      Handle  HelpFile  IniPath  InstanceMode  KeepOnClos
      e  LastError  Locale  LoginTimeout  Properties  Que
      ueEvents  SystemDB  Type  TypeList  Version  Worksp
      aces
```

Pre-Version 11 Behaviour

In previous versions of Dyalog APL, you could obtain this information from the PropList, MethodList and EventList properties of the object. Note that these 3 properties are internally generated by Dyalog APL and are not exported by the object itself. You could also obtain this information by executing the system commands `)PROPS`, `)METHODS` and `)EVENTS` inside the namespace that is associated with an instance of the object.

For backwards compatibility, these capabilities are retained when `⎕WX` is 0 or 1.

For example:

```
      ⎕WX←1
      'DB'⎕WC'OleClient' 'DAO.DBEngine.36'
      )CS DB
#.DB
      )METHODS
BeginTrans      CommitTrans     CompactDatabase
CreateDatabase  CreateWorkspace Idle    ISAMStats
OpenConnection  OpenDatabase    RegisterDatabase
RepairDatabase  Rollback        SetOption

      )PROPS
AutoBrowse      ChildList       ClassID ClassName
Data    DefaultPassword DefaultType     DefaultUser
Errors  Event   EventList       Handle  HelpFile
IniPath InstanceMode    KeepOnClose     LastError
Locale  LoginTimeout    MethodList      Properties
PropList        QueueEvents     SystemDB        Type
TypeList        Version Workspaces
```

Or, for example, using an ActiveX Control:

```
      NAME←'Microsoft Office Chart 9.0'
      'MOC'⎕WC'OCXClass'NAME
      'F' ⎕WC'Form'
      'F.MOC' ⎕WC 'MOC' ⍝ Instance of MOC
      )CS F.MOC
#.F.MOC
      )PROPS
Attach  AutoConf        Border  BuildNumber     CanUndo
Charts  ChartDataSources        ChartLayout
ChartSpaceLegend        ChartSpaceTitle ChartWrapCount
ChildList       ClassID ClassName       Constants...
```

Using the Property Sheet

The simplest way to obtain further information about an OLE property, method or event is to display its Property Sheet.

To do this, change space to the namespace that represents the object, type the name (or place the cursor over the name) of the property, method or event in question, press the right mouse button and select Properties from the context menu.

The information displayed for the OpenDatabase method that is provided by the DAO.DBEngine OLE object is shown below.
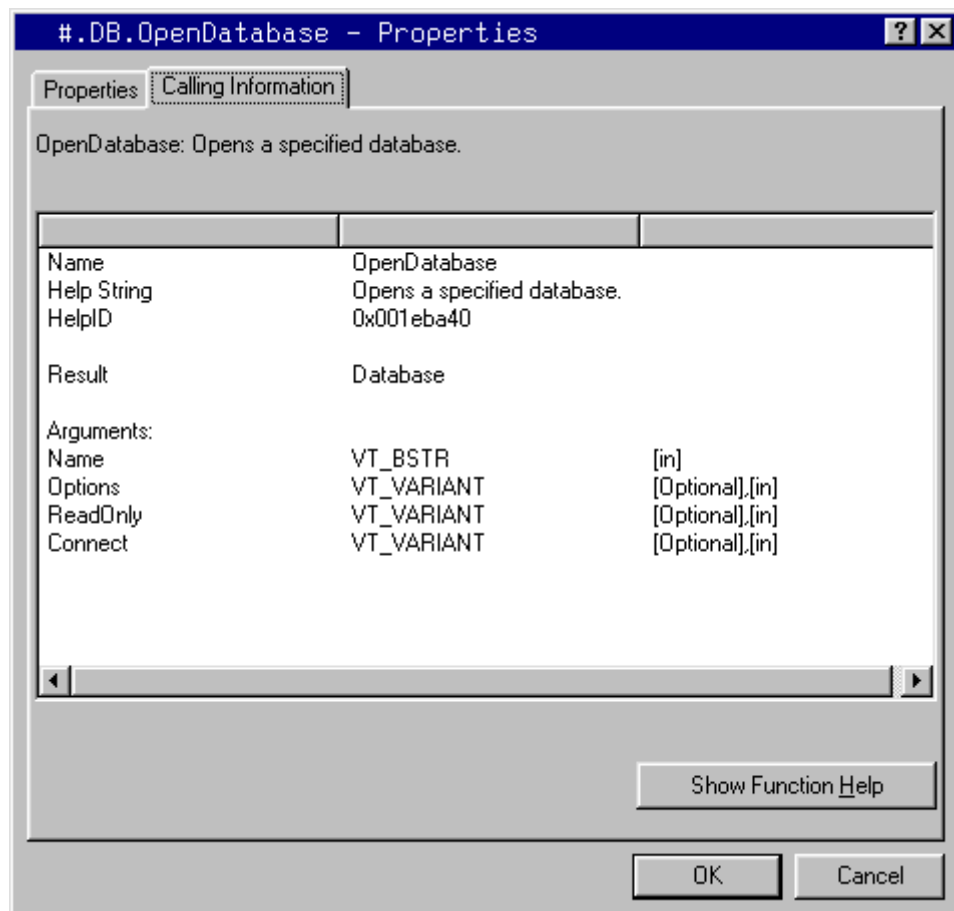
Using the Workspace Explorer

You can also obtain information using the Workspace Explorer.

If you have created an instance of an object, you can navigate to it using the Explorer and then browse its Events, Methods and Properties. The picture below illustrates the effect of browsing the object DB that is connected to DAO.DBEngine.35.
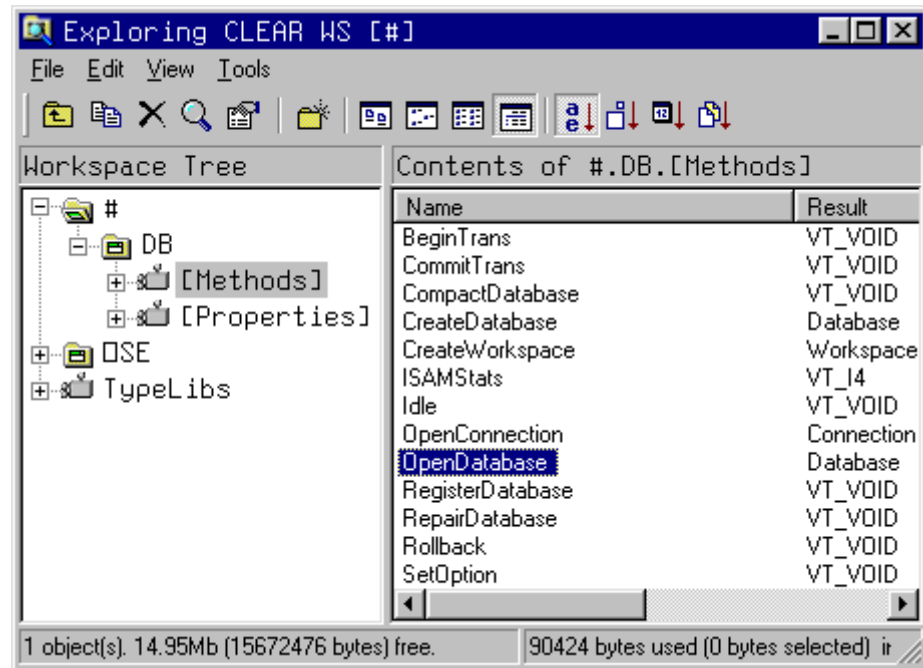
To obtain detailed information about a specific property, event, or method, just open the appropriate folder and select the name you want. The details will be displayed in the list view pane.

The same information can be obtained by browsing the *Loaded Libraries* folder. This folder will be displayed if the *View/Type Libraries* menu item is checked and the appropriate library has been loaded. The library will be loaded if you have ever created an instance of the object in this workspace. Alternatively, you may navigate to the information using the *Registered Library* folder.
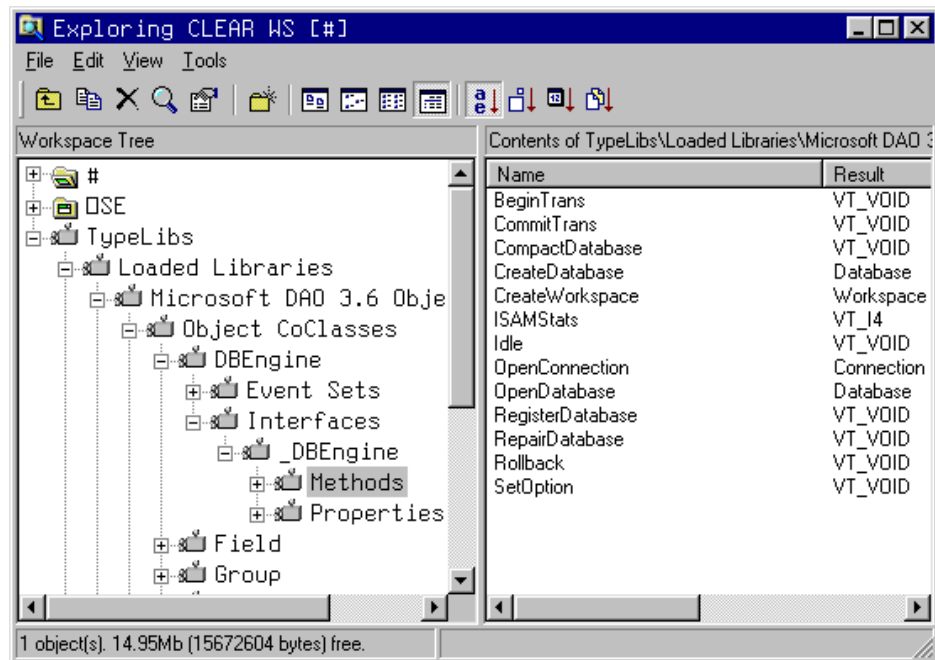
GetPropertyInfo Method

You can also obtain information about the properties exposed by a COM object, using the GetPropertyInfo method. Note that this is a Dyalog APL method, added to the object, and not a native method provided by the object itself.

For example, the DAO.DBEngine OLE object exposes a property called Version. You can discover the meaning of the Version property as follows:

```
      GetPropertyInfo 'Version'
Returns the version of the database engine.  VT_BSTR
```

Or, using ⎕NQ

```
      +2 ⎕NQ ''  'GetPropertyInfo' 'Version'
Returns the version of the database engine.  VT_BSTR
```

This tells you that the property value is a character string (VT_BSTR) that contains the version number of the database engine.

```
      Version
3.51
```

GetMethodInfo Method

You can also obtain information about the methods exposed by an OLE object, using the GetMethodInfo method. Note that this is a Dyalog APL method, added to the object, and not a native method provided by the object itself.

For example, the DAO.DBEngine OLE object exposes a method called OpenDatabase. You can obtain information about the OpenDatabase method as follows:

```
      ↑GetMethodInfo 'OpenDatabase'
 Opens a specified database  VT_DISPATCH
 Name                        VT_BSTR
 [Exclusive]                 VT_VARIANT
 [ReadOnly]                  VT_VARIANT
 [Connect]                   VT_VARIANT
```

This tells you that the method opens a specified database and that the result is of type VT_DISPATCH. Furthermore, the function takes up to four arguments, the first of which (called Name) is a character string (VT_BSTR). The remaining 3 arguments (called Exclusive, ReadOnly and Connect) are optional (their names are surrounded by []) and of type VT_VARIANT.

GetEventInfo Method

Let's use the Windows Media Player as an example. First we must load the Control by creating an OCXClass object using ⎕NEW.

```
  wmp←⎕NEW'OCXClass'(⊂'ClassName' 'Windows Media Player')
  'f'⎕WC'Form'
  'f.wmp'⎕WC'wmp'
```

Next we can find out what events it supports using ⎕NL ¯8.

```
      wmp.⎕NL ¯8
Buffering  Click  DblClick  Disconnect  DisplayModeChange
     DVDNotify  EndOfStream  Error  KeyDown  KeyPress
    KeyUp  MarkerHit  MouseDown  MouseMove  MouseUp  New
   Stream  OpenStateChange  PlayStateChange  PositionCh
    ange  ReadyStateChange  ScriptCommand  Warning
```

Then, we can obtain information about a particular event (or events) by invoking a GetEventInfo method. Note that in the case of the Windows Media Control it is necessary to query the instance of the control (f.wmp) as opposed to the instance of the OCXClass (wmp). For example, you can ask it about its MouseDown event. The result is a vector, each element of which is a 2-element vector of character vectors.

```
      ρINFO←⊂f.wmp.GetEventInfo'MouseDown'
5
```

The first element contains a description of the event and the data type of its result (few events generate results, so this is usually VT_VOID), i.e.

```
      ⊃INFO
Sent when a mouse button is pressed  VT_VOID
```

Subsequent elements describe the name and data type of each of the parameters to the event. These are the items that will appear as the third and subsequent elements of the event message that is passed as the right argument to a callback function or returned as the result of ⎕DQ. In this case:

```
      ↑1↓INFO
 Button      VT_I2
 ShiftState  VT_I2
 x           VT_COORD
 y           VT_COORD
```

This information tells us that the first parameter *Button* is a 2-byte integer value which (presumably) is the number of the mouse button that the user has pressed. The second parameter *Shift* is also a 2-byte integer and (presumably) reports the keyboard shift state. The third and fourth parameters *X* and *Y* are of data type VT_COORD.

Obtaining On-line Help

You can display the help topic associated with a property, method, or event by selecting Help from its context menu or using the help button in its property sheet.

Note that the name of the object's help file is provided by its HelpFile property.

For example, in the case of the DAO.DBEngine OLE object:

```
      ⎕WG'HelpFile'
C:\PROGRAM FILES\COMMON FILES\MICROSOFT SHARED\DAO\dao.hlp
```

For Office 2000 applications, you will need to install the MSDN to obtain the appropriate help files.

Methods

When you create an instance of a COM object, the methods and the properties are directly accessible from the corresponding namespace.

Calling Methods

You invoke a method in an OLE object as if it were an APL function in your workspace.

If a method takes no parameters, you must invoke it as if it were niladic.

If a method takes parameters, you must call it as if it were monadic. Each element of its argument corresponds to each of the method's parameters.

If a method takes a parameter declared as a string (VT_BSTR) you must call it with an **enclosed** character vector.

Note: In previous versions of Dyalog APL, a character vector was automatically enclosed if required. For backwards compatibility you may select old or new behaviour using ⎕WX. If ⎕WX is 3 (the default) you **must** enclose a single string argument. IF ⎕WX is 0 or 1, you **may** supply a simple character vector.

For example, the OpenDatabase method in the DAO.DBEngine OLE server may be called with a single parameter that specifies the name of the database to be opened. You may call it from APL with either of the following two expressions:

```
OpenDatabase 'c:\example.mdb' ⍝only if ⎕WX is 0 or 1
OpenDatabase ⊂'c:\example.mdb'⍝any value of ⎕WX
```

Arrays and Pointers

Many parameters to OLE methods are specified by pointers. If, for example, the parameter type is VT_BSTR, it means that the calling routine must supply a pointer to (i.e. the address of) a character string.

Similarly, if the parameter type is defined to be VT_VARIANT, it means that the parameter is the address of an arbitrary array (the VT_VARIANT data type actually maps nicely onto a Dyalog APL nested array).

The rule is that if a pointer is required, APL will provide it automatically; you do not have to do so. Instead, all you do is supply the value.

Optional Parameters

Methods are often defined to have optional parameters. For example the parameters defined for the OpenDatabase method provided by the DAO.DBEngine OLE object are:

```
Name         VT_BSTR
[Exclusive]  VT_VARIANT
[ReadOnly]   VT_VARIANT
[Connect]    VT_VARIANT
```

To call the corresponding APL function, you may supply a nested array that contains 1,2, 3or 4 elements corresponding to these parameters.

The parameters to some methods are all optional. This means that the method may be called with or without any parameters. As APL does not support this type of syntax, the special value θ (zilde) is used to mean "0 parameters".

For example, the parameters for the Idle method provided by DAO.DBEngine are defined to be:

```
[Action]  VT_VARIANT
```

This means that the method takes either no arguments or one argument. To call it with no argument, you must use θ (zilde), for example:

```
    Idle θ
```

Note that you cannot therefore call a function in an *APL* server with a single argument that is an empty numeric vector.

Output Parameters

You may encounter parameters whose data type is defined explicitly as a pointer to something else, for example VT_PTR to VT_UI4 specifies a pointer to an unsigned 4-byte integer.

In these cases, it usually means that the calling routine is expected to pass an address into which the OLE method will place a value.

When you invoke the method you must use data of the type pointed to.

The result of the method is then a vector containing the result defined for the method, followed by the (new) values of the output parameters. This is similar to the mechanism used by ⎕NA.

Named Parameters

Visual Basic syntax allows you to specify parameters by position or by name; rather like ⎕WC and ⎕WS. For example the parameters defined for the OpenDatabase method provided by the DAO.DBEngine OLE object are:

```
Name           VT_BSTR
[Exclusive]    VT_VARIANT
[ReadOnly]     VT_VARIANT
[Connect]      VT_VARIANT
```

You could call this method from Visual Basic using the syntax:

```
Set Db = OpenDatabase(Name:="c:\example.mdb",_
                      ReadOnly:=True)
```

You may do the same thing from Dyalog APL, using ⎕WS syntax. For example, the equivalent call from APL would be:

```
OpenDatabase('Name' 'c:\example.mdb')('ReadOnly' 1)
```

Note that you may only use named parameters if they are supported by the method. Many methods do not allow them.

Methods that return Objects

Object hierarchies in OLE are not static, but are created dynamically by calling methods that return objects as their result.

If the data type of the result of a method is a pre-defined object type, or VT_DISPATCH or VT_COCLASS, or VT_PTR to VT_DISPATCH or VT_PTR to VT_COCLASS, the result returned to APL is a *namespace*. If the result is assigned to a name, the value associated with that name becomes a *namespace reference*. For example, GetMethodInfo tells us that the syntax for the OpenDatabase method provided by the OLE object DAO.DBEngine is as follows:

```
      ↑ DB.GetMethodInfo 'OpenDatabase'
 Opens a specified database  VT_DISPATCH
 Name                        VT_BSTR
 [Exclusive]                 VT_VARIANT
 [ReadOnly]                  VT_VARIANT
 [Connect]                   VT_VARIANT
```

The data-type of the result is VT_DISPATCH, so it returns an object; indeed the help for the method tells us that it returns a Database object. The function could be called from APL as follows:

```
      DB←OpenDatabase ⊂'example.mdb'
```

Alternatively, you may simply use the result as an argument to a defined function or as the argument to ⎕CS or :With, thereby switching into the namespace returned by the method. For example:

```
      :With OpenDatabase ⊂'example.mdb'
         ...
      :EndWith
```

Notice that in both these cases, the namespace associated with the result of the OpenDatabase method is *unnamed*. Assigning the result of OpenDatabase to DB does not set the namespace *name* to DB, it merely assigns a namespace *reference* to DB.

To preserve compatibility with previous versions of Dyalog APL that did not support namespace references, a method that returns an object may be called with the name of the (new) namespace as its left argument. Note that OLE methods do not themselves accept left arguments, so this extension does not conflict with OLE conventions.

```
      'DB' OpenDatabase ⊂'example.mdb'
```

This expression creates a new namespace called DB associated with a new object in the OLE Server. Note that if you invoke the OpenDataBase method in this way, its result is a number that represents the *Dispatch Interface* of the new object. This is done to preserve compatibility with previous versions of Dyalog APL.

Properties

By default, Properties exposed by a COM object behave in the same way as Properties exposed by Dyalog APL Classes.

To query the value of a property, you simply reference it. To set the value of the property, you assign a new value to it. If the Property is an Indexed Property, you may use indexing to set or retrieve the value of a particular element.

Note that in previous versions of Dyalog APL, indexed Properties of COM objects were exposed as Methods and for backwards compatibility this behaviour may be retained by setting ⎕WX to 0 or 1 (the default value is 3). See Language Reference.

If the old (pre-Version 11.0) behaviour is selected., indexed properties are exposed as methods and you treat the property as if it were an APL function. To obtain the value of the property, you must call it monadically, specifying the required index (or other information) as the argument. To set the value of the property, you must call it dyadically, specifying the required index (or other information) as the right argument and the new value as the left argument.

The data type of the variable is reported by the `GetPropertyInfo` method. Conversion between APL data types and OLE data types is performed automatically.

If you attempt to set the value of a property to an something with an inappropriate data type, APL will generate a `DOMAIN ERROR`.

If you set the value to something of the correct data type, APL will pass it through the OLE interface. However, the OLE object may itself reject the new value. In this case, APL will also generate a `DOMAIN ERROR`. However, the OLE error information may be obtained from the `LastError` property of the object or Root. The error is also displayed in the Status Window.

Note that if ⎕WX is 0 or 1, `)PROPS` and PropList report the names of all of the properties of an object, regardless of whether the property is implemented as a variable or as a function. You can tell whether or not the property takes an argument (and therefore behaves as a function) from its property sheet, using GetPropertyInfo, or from the documentation for the object in question.

Properties as Objects

Dyalog APL permits an object hierarchy to be represented by a namespace hierarchy. In other words, the relationship between one object and another is a parent-child relationship whereby one object owns and contains another.

Visual Basic has no such mechanism and the relationship between objects has to be specified in another way. This is commonly done using properties. For example, an object view of a bicycle could be a hierarchy consisting of a bicycle object that contains a Frame object, a FrontWheel object and a RearWheel object. In Visual Basic, you could represent this hierarchy as a Bicycle object having Frame, FrontWheel and RearWheel *properties* which are (in effect) pointers to the sub-objects. The properties are effectively used to tie the objects together.

An extension of this idea is the Visual Basic Collection object. This is a special type of object, that is somewhat similar to an array. It is used where one object may contain several objects of the same type. For example, a Wheel object could contain a Spokes collection object which itself contains a number of individual Spoke objects. Collection objects are usually implemented as properties.

When you reference the value of an object property, you will get a namespace.

Using the bicycle analogy, you could recreate the object hierarchy in the APL workspace as follows:

```
'BIKE'  □WC'OLEClient' 'EG.Bicycle'
FRONT ← BIKE.FrontWheel
REAR ← BIKE.RearWheel
```

The result would be three namespaces, one named `BIKE`, and two unnamed namespaces referenced by `FRONT` and `REAR`. Each contains the specific properties, methods and events exposed by the three corresponding objects.

Note however, that in this example `BIKE`, `FRONT` and `REAR` are all top-level namespaces; a proper parent/child representation can be achieved by making `FRONT` and `REAR` child namespaces of `BIKE`, for example:

```
BIKE.FRONT ← BIKE.FrontWheel
BIKE.REAR ← BIKE.RearWheel
```

or

```
:With BIKE
    FRONT ← FrontWheel
    REAR ← RearWheel
:EndWith
```

This example illustrates that when you work with an OLE object, you have a choice whether to represent an object hierarchy as a namespace *tree* or just as a collection of otherwise unrelated namespaces.

Events

Events generated by OLE objects are provided via an *event sink* which is simply an interface that defines a collection of events that may be generated by the object. Objects may support more than one event sink and may or may not define them in a type library. By default, events generated by COM objects are processed like all other events in Dyalog APL.

This means that if you attach a callback function to an event in an instance of an OCXClass object, the events are queued up when they are received and then processed one-by-one, by `⎕DQ`, from the internal queue. This is the mechanism used to process all events in Dyalog APL and it has many advantages:

- Events are handled in an orderly manner

- An event cannot interrupt a callback that is processing a previous event

- Incoming events are held up so that you can trace a callback function

The disadvantage of this approach is that, for internal reasons, your APL callback function is unable to return a result to the ActiveX control, or to modify any of the arguments supplied by the event. This is a severe problem if the COM object relies on callbacks to control certain aspects of its functionality.

The QueueEvents property allows you to change the normal behaviour so that it is possible for a callback function to return a result to a COM object.

If QueueEvents is 1, which is the default, the result (if any) of your callback function is not passed back to the COM object but is discarded. Thus you cannot, for example, inhibit or modify the default processing of the event by the COM object.

If instead you set QueueEvents to 0, the callback function attached to the event is executed immediately, even if there are other APL events before it in the internal event queue. The result of your callback function is then passed back to the COM object which may use it to inhibit or modify its normal event processing.

See QueueEvents for further details.

Using the Microsoft Jet Database Engine

The SQL function in workspace `samples\ole\oleauto.dws` is a simple example showing how you can call the Microsoft Jet database engine using OLE.

SQL is dyadic. The left argument is the path-name of an Access database; the right argument is a query in the form of an SQL statement.

The result is a matrix containing the records that match the query.

For example:

```
      FILE← 'c:\Program Files\Microsoft
Office\Office\Samples\Northwind'
      QUERY←'Select * From Suppliers'
     ρFILE SQL QUERY
29 11
```

The SQL Function

```
     ∇ DATA←DATABASE SQL QUERY;DB;DBS;RCS
[1]   ⍝ Uses the OLE Server DAO.DBEngine to perform a
[2]   ⍝ query on an MS Access database
[3]
[4]     'DB'⎕WC'OleClient' 'DAO.DBEngine.35'
[5]
[6]    :Trap 11 ERROR
[7]        DBS←DB.OpenDatabase⊂DATABASE
[8]        RCS←DBS.OpenRecordset⊂QUERY
[9]        DATA←�checkRCS.GetRows 999
[10]   :Else
[11]       DATA←'DB'⎕WG'LastError'
[12]   :EndTrap
     ∇
```

Let us examine how the function works.

```
    [4]    'DB'⎕WC'OleClient' 'DAO.DBEngine.35'
```

This statement creates a new namespace called `DB` that is connected to the DAO.DBEngine OLE Server. After the statement has executed, `DB` is essentially an instance of the object and exposes the methods and properties provided by the object.

```
    [7]    DBS←DB.OpenDatabase⊂DATABASE
```

The `OpenDatabase` method is called with the name of the database file as its single argument (other parameters are optional and omitted here). The result of `OpenDatabase` is a (new) Database object whose namespace reference is assigned to `DBS`.

```
[8]    RCS←DBS.OpenRecordset QUERY
```

The `OpenRecordset` method is called with a character vector containing an SQL statement as its argument. The result of `OpenRecordset` is a (new) Recordset object whose namespace reference is assigned to `RCS`.

```
[9]    DATA←⍉RCS.GetRows 999
```

The `GetRows` method takes a single parameter which is the number of rows to be fetched. This simple example ignores the possibility that there may be more than 999 records to be fetched and ignores the possibility of `WS FULL`. The result is a nested matrix containing the data. In this case, the data is transposed.

It is not actually necessary to assign the results of the expressions in lines [7] and [8]. These expressions, which return namespaces, can simply be parenthesised and the entire query can be executed in a single statement as illustrated by function `SQL1`.

```
      ∇ DATA←DATABASE SQL1 QUERY;DB
[1]    ⍝ Shorter version of SQL
[2]
[3]    'DB'⎕WC'OleClient' 'DAO.DBEngine.35'
[4]
[5]    :Trap 11
[6]        DATA←⍉((DB.OpenDatabase⊂DATABASE).
   OpenRecordset⊂QUERY).GetRows 999
[7]    :Else
[8]        DATA←DB.LastError
[9]    :EndTrap
      ∇
```

OLE Objects without Type Information

If you create an instance of a COM object that does not provide Type Information, the resulting namespace be empty and will appear to provide no methods and properties.

Nevertheless, in most cases, you will still be able to access its methods and properties using *Late Binding* or SetMethodInfo and SetPropertyInfo as follows.

Late Binding

Late Binding in this context means that the association between an APL name and a method or property exported by the COM object is deferred until the name is used.

If you refer to a name inside the OLEClient namespace that would otherwise generate a VALUE ERROR, APL asks the COM object if it has a member (method or property) of that name.

The mechanism permits APL to determine only that the member is exported; it says nothing about its type (method or property) nor its syntax. If the response from the COM object is positive, APL therefore makes the most general assumption possible, namely:

- That the member is a method
- That it may take up to 16 optional arguments
- That each argument is input/output (i.e. specified via a pointer)
- That the method returns a result.

This means that if you know, from its documentation or another source, that a COM object provides a certain Method or Property, you may therefore access that member by simply calling a function of that name in the OLEClient namespace. Note that any parameters you pass will be returned in the result, because APL assumes that all parameters are input/output. Furthermore, APL will be unable to check the validity of the parameters you specify because it does not know what data types are expected.

SetMethodInfo and SetPropertyInfo

The SetMethodInfo and SetPropertyInfo methods provide a mechanism for you to precisely specify the missing Type Information for the methods and properties that you wish to use. See *Object Reference* for further details.

Note that whether you use late binding or SetMethodInfo/SetPropertyInfo, any sub-object namespaces that you create by invoking the methods and properties in the top-level object, will also have no visible methods and properties. Therefore, if the Type Information is missing, Late Binding or SetMethodInfo and SetPropertyInfo must be used to access all the methods and properties that you wish to use, wherever they occur in the object hierarchy.

Events

When type library information is available, Dyalog APL automatically connects the appropriate event sinks and establishes the EventList property for the object when it is created. However, if the COM object does not declare its event sinks in a type library, it is necessary to connect to them manually. To support these cases, the following methods are used. These apply to top-level COM objects and to the namespaces associated with any other COM objects exposed by them.

Method	Description
OLEListEventSinks	Returns the names of any event sinks currently attached to an object. An event sink is a set of events grouped (for convenience) by a COM object.
OLEAddEventSink	Attaches the namespace associated with an object to a specific event sink that it supports. If successful, new event names will appear in the EventList property of the namespace. This is the only way to access events from an event sink that is not described in the object's Type Information.
OLEDeleteEventSink	Removes the events associated with a particular event sink from the EventList property of the namespace associated with an object.

Collections

A collection is a special type of object that represents a set of other objects. Collections are typically implemented as properties. For example, the Excel Sheet object has a property named Sheets whose value is a collection object that represents a set of worksheets. Collections typically have a property called Count, which tells you how many objects there are, and a Default Property named Item that provides access to each member of the set. Item typically accepts a number or a name as an index and returns a reference to an object.

For example, if a workbook contains two worksheets named "P&L" and "2002 Sales" respectively, they might be accessed as follows:

```
      S1←Sheets.Item[1]
      S1.Name
P&L
      S2←Sheets.Item ['2002 Sales']
      S2.Index
2
```

Note that in old versions of Dyalog APL (pre-Version 11.0) the Item property was exposed as a method. This old behaviour may be select by setting ⎕WX to 0 or 1 when you create the object. In which case:

```
      S1←Sheets.Item 1
      S1.Name
P&L
      S2←Sheets.Item '2002 Sales'
      S2.Index
2
```

Note that some collections work in origin 0 and some in origin 1; there is no way to tell which applies except from the documentation. Furthermore, collections are used for all sorts of purposes, and may not necessarily permit the instantiation of more than one member of the set at the same time. Collections are *not* the same as arrays.

As mentioned above, the Item property is typically the Default Property (see Language reference) of a Collection, so indexing may be applied directly to the Collection object.

```
      Sheets[1 2].Name
 P&L  2002 Sales
```

The `:For - :EndFor` control structure provides a convenient way to enumerate through the members of a collection without using the Item property. For example, the following code snippet accumulates the values in an Excel worksheet collection.

```
DATA←0ρ⊂0 0ρ0
:For S :In Sheets  ⍝ Enumerate SHEETS collection
    DATA,←⊂S.UsedRange.Value2
:EndFor
```

Null Values

COM methods and properties frequently return null values for which there is no direct equivalent in the APL language. Instead, the system constant `⎕NULL` is used to represent a null value.

The following spreadsheet contains a number of empty cells.

Using the Excel.Application COM object, the contents of the spreadsheet can be obtained as follows:

```
      'EX'⎕WC'OLEClient' 'Excel.Application'
      WB←EX.Workbooks.Open 'simple.xls'

      WB.Sheets[1].UsedRange.Value2
[Null]  [Null]  [Null]  [Null]  [Null]
[Null]    Year  [Null]  [Null]  [Null]
[Null]    1999    2000    2001    2002
[Null]  [Null]  [Null]  [Null]  [Null]
Sales      100      76     120     150
[Null]  [Null]  [Null]  [Null]  [Null]
Costs       80      60     100     110
[Null]  [Null]  [Null]  [Null]  [Null]
Margin      20      16      20      40
```

To determine which of the cells are filled, you can compare the array with ⎕NULL.

```
      ⎕NULL≠¨WB.Sheets[1].UsedRange.Value2
0 0 0 0 0
0 1 0 0 0
0 1 1 1 1
0 0 0 0 0
1 1 1 1 1
0 0 0 0 0
1 1 1 1 1
0 0 0 0 0
1 1 1 1 1
```

⎕NULL should also set the values of COM properties to null.

Additional Interfaces

Most COM objects and their sub-objects provide information about their methods and properties through the IDispatch interface which is the normal interface used for OLE Automation. When you create an instance of an OLEClient object or an OCXClass object, Dyalog APL uses this interface to gain the information it requires.

If an object does not provide an IDispatch interface, or if an object provides additional functionality through other interfaces, it is possible to access the object's functionality using the OLEQueryInterface method.

In addition, if an object exposes sub-objects using an interface other than IDispatch, you may access these sub-objects using the OLEQueryInterface method..

See OLEQueryInterface for further details.

Writing Classes based on OLEClient

You may define APL Classes (See Language Reference) based upon the OLEClient object. For example:

```
:Class Excel: 'OLEClient'
    ∇ ctor wkbk
      :Access Public
      :Implements Constructor :Base ,⊂('ClassName'
'Excel.Application')
      Workbooks.Open ⊂wkbk
    ∇
:EndClass ⍝ Excel


      XL←⎕NEW Excel 'f:\help11.0\days.xls'
      XL.Workbooks[1].Sheets[1].UsedRange.Value2
  From      To    Days  Hours
 38790   38791       0   3.25
 38792   38792  [Null]   2.25
 38793   38793  [Null]   2.5
 38799   38799  [Null]   5
 38800   38800  [Null]   3
[Null]  [Null]  [Null]  16
```

C H A P T E R 12

OLE Automation Server

Introduction

OLE Automation allows you to drive one application from another and to mix code written in different programming languages. In practical terms, this means that you may write a subroutine to perform calculations in (say) C++ and use the subroutine directly in Visual Basic or Excel. Equally, you could write the code in Visual Basic and call it from C++. Dyalog APL/W is a fully subscribed member of this code-sharing club.

OLE Automation is, however, much more than just a mechanism to facilitate cross-application macros because it deals not just with subroutine calls but with *objects*. An object is a combination of code and data that can be treated as a unit. Without getting too deeply into the terminology, an object defines a *class*; when you work with an object you create one or more *instances* of that class.

OLE objects are represented in Dyalog APL by namespaces.

This chapter describes how you can write an OLE Automation Server in Dyalog APL.

Namespaces and Objects

There is a direct correspondence between the object model and Dyalog APL namespace technology, a correspondence that is thoroughly exploited in the implementation of OLE Automation.

An OLE object is simply a collection of methods (code that performs tasks) and properties (data that affects behaviour). An object corresponds directly to a Dyalog APL namespace which contains functions that do things and variables that affect things. Furthermore, OLE objects are hierarchical in nature; objects may contain sub-objects just as namespaces may contain sub-namespaces. To complete the picture, an OLE Server is an application that provides (exposes) one or more OLE objects. Thus an OLE Server corresponds directly to a workspace that contains one or more namespaces.

However, when you access an OLE object, you do so by creating an *instance* of its class and you may work with several instances at the same time. Furthermore, several applications may access the same OLE object at the same time, each with its own set of instances. Each instance inherits its methods (functions) and the initial values of its properties from the class. However, different property values will soon be established in different instances so they must be maintained separately.

Dyalog APL/W includes the capability for a namespace to spawn instances of itself. Initially, a new instance is simply a pointer to the original namespace (not a copy), but as soon as anything in it is changed, the new value is recorded separately. Thus instance namespaces will typically share functions but maintain separate sets of data.

Writing an APL OLE Server

The following steps are required to create an OLE Automation Server in Dyalog APL/W:

1. Create a workspace containing an OLEServer namespace. This namespace represents an OLE Object and may contain as many functions and variables as you want to provide the functionality you require. It may also contain other OLEServer namespaces to represent sub-objects in an object hierarchy.

2. For each of the functions and variables that you wish to expose as methods and properties of your object, you must declare the data types of their parameters and results. You can do this manually, using the COM Properties tab of the Object Properties dialog box, or by invoking the SetFnInfo and SetVarInfo methods. Note that non-exported functions and variables, sub-namespaces and defined operators may be used internally, but are not available directly to an OLE Automation client. It is also possible to generate events from an OLEServer. The mechanism is the same as for an ActiveXControl and is described in the next chapter.

3. Select *Export* from the Session *File* menu and choose in-process or out-of-process COM Server as you prefer.

Rules for Exported Functions

There are certain fundamental differences between OLE syntax and APL syntax.

For example, OLE methods may take any number of arguments whereas APL is confined to two; a left and a right.

Secondly, some of the arguments or even *all* of the arguments to an OLE method may be optional. You cannot however call a monadic APL function with no arguments; in APL there is a clear distinction between niladic functions and functions that take an argument.

Furthermore, the number and type of the arguments for each OLE method must be registered in advance so that OLE knows how to call it.

These factors mean that certain rules must be adopted so that APL can register your APL functions as OLE methods.

Rule 1: Exported APL functions must be niladic or monadic *defined* functions; dyadic functions, dynamic functions, derived functions and operators are not allowed. However, ambivalent functions may be called (monadically) by OLE.

Rule 2: Character arrays whose rank is greater than 1 are passed as 1-byte integer arrays. This means that 1-byte integer matrices and higher-order arrays will always be converted to character arrays.

Rule 3: An exported APL function may not be called with an empty numeric vector (zilde) as its single argument. Zilde is used by an APL client to call a non-niladic OLE method with no arguments.

Rule 4: If an exported APL function is called with more than one parameter, its argument will be a nested vector. If it is called with a single parameter that is a character vector or an array whose rank is greater than 1, the argument supplied to the APL function will be a simple array. Effectively, a 1-element nested array received from an OLE Client is disclosed.

Out-of-Process and In-Process OLE Servers

Dyalog APL allows you to create both out-of-process OLE Servers and *in-process* OLE Servers. An *out-of-process* OLE Server runs as a completely separate Windows program that communicates with one or more client programs. An *in-process* OLE Server is implemented as a Dynamic Link Library (DLL) that is loaded into the client process and becomes part of its address space.

The main **advantage** of an *in-process* OLE Server is that communication between the client application and the OLE Server is fast. Communication between clients and *out-of-process* OLE Servers has to go through a separate OLE layer in Windows that incurs a certain overhead. Another **advantage** is that *in-process* OLE Servers are simpler to administer and simpler to install.
The main **disadvantages** of *in-process* OLE Servers is that there can only be one client per server and they do not support DCOM directly.

ClassID, TypeLibID and other properties

Windows COM objects are identified using a system of *Globally Unique Identifiers* (GUIDs). When you create an OLEServer object using ⎕WC, APL creates a number of GUIDs and allocates them to the OLE Server. One of these is a *Class Identifier* (often abbreviated to CLSID) that will uniquely identify your OLE object. This is stored in the ClassID property of the OLEServer. Another GUID identifies the Dispatch interface of the object but is not available via a property.

An out-of-process COM server requires a separate Type Library file. This is a binary file that describes the methods (functions) and properties (variables) exposed by the OLEServer namespace(s) in the workspace. The Type Library is identified by a GUID and by its file name. The file name (which is constructed from the workspace name with a .TLB extension) is stored in the TypeLibFile property of the OLEServer namespace. The GUID is generated when it is first needed and is stored in the TypeLibFileID property of the OLEServer namespace. Note that if the workspace contains several OLEServer objects, their TypeLibFile and TypelLibID properties all have the same values.

Note that the equivalent information for an in-process OLE Server is bound into the DLL and not stored in a separate file.

In-process OLE Servers

Exporting

When you use File/Export to create an *in-process* OLE Server, the following steps are performed.

APL first saves your workspace to a *temporary* file. Then it creates a *temporary* Type Library File that describes each of the OLEServer objects in the workspace. Next, it creates a Dynamic Link Library (DLL) file (whose name defaults to the name of your workspace but with a .DLL extension) by merging the workspace saved in the temporary file with the file OCXSTUB.DLL. Finally, it registers your OLE Server by updating the Windows Registry. Your OLE Server DLL is self-contained and is independent of your workspace. The temporary files are then deleted.

Execution

In-process OLEServers are hosted (executed) by the Dyalog APL DLL. If you export your OLE Server with *Runtime application* checked, it will be bound with the run-time version, If this checkbox is cleared, your OLE Server will be bound by the development version.

If an in-process OLE Server, that is bound with the run-time Dyalog APL DLL generates an untrapped error, an OLE Automation error will be reported.

If an in-process OLE Server, that is bound with the development Dyalog APL DLL generates an untrapped error, the APL Session will appear and you can use it to debug the problem and continue. Note that at this point, the development DLL will load your Session file so that all of your session tools are available during debugging. If your Session file runs any initialisation code that references external files, remember that this code will be executed in the current working directory of the host process.

For further details, see *User Guide, Chapter 2*.

Registering and Unregistering

During development, an in-process OLE Server is automatically registered when you create it using *File/Export*.

The Windows utility REGSVR32.EXE should be used to register an *in-process* OLE Server independently, or to install a runtime in-process OLE Server on a target computer. For example:

```
C:\Dyalog101>regsvr32 mysvr.dll
```

REGSVR32 should also be used (with the */u* flag) to un-register an in-process OLE Server. For example:

```
C:\Dyalog101>regsvr32 /u mysvr.dll
```

Note that in both cases, REGSVR32 actually starts the OLE Server. This in turn loads the appropriate Dyalog APL DLL. If you are using the development DLL, note that if your session start-up code fails for any reason, the REGSVR32 process will hang and have to be terminated using the Task Manager.

Out-of-process OLE Servers

Exporting

When you use File/Export to create an *out-of-process* OLE Server, the following steps are performed.

APL first creates a single Type Library File that describes all of the OLEServer objects in the workspace. It then registers your OLE Server by updating the Windows Registry with, among other things, the names and ClassIDs of your workspace and Type Library file.

Note that the type information is taken from your active workspace and not the saved workspace. It is up to you to ensure that your saved workspace (which will actually be used when the OLE Server is invoked) is kept in step.

For example, if you were subsequently to remove the OLEServer objects from your workspace and re-save it, or save a completely different workspace with the same pathname, your OLE Server would fail to start because the Type Library and Registry and no longer synchronised with your workspace.

Execution

An out-of-process OLE Server is implemented by a separate Dyalog APL process (DYALOG.EXE or DYALOGRT.EXE) that loads your workspace when it starts.

If an out-of-process OLE Server, that is bound with the *run-time* Dyalog APL program, generates an untrapped error, an OLE Automation error will be reported.

If an out-of-process OLE Server, that is bound with the *development* Dyalog APL program, generates an untrapped error, the APL Session will appear and you can use it to debug the problem and continue. In previous versions of Dyalog APL, the visibility of the APL Session for debugging was controlled by the ShowSession property. Setting ShowSession to 1 will cause the Session to be displayed immediately, when the OLE Server is started. However, setting ShowSession to 0 will not prevent the Session from appearing if an untrapped APL error occurs.

Registering and Unregistering

During development, an out-of-process OLE Server is automatically registered when you create it using *File/Export*.

An out-of-process OLEServer may also be registered by calling its OLERegister method. This performs the same tasks as *File/Export*, but without any user-interaction.

OLERegister is the recommended way to install an *out-of-process* OLEServer on a target computer as a run-time application.

An out-of-process OLEServer may be unregistered by calling its OLEUnRegister method.

Registry Entries

This section describes the entries that are written into the Windows Registry when APL registers an *out-of-process* OLEServer.

All registry entries are written as sub-keys of the primary key `HKEY_LOCAL_MACHINE\SOFTWARE\Classes` of which `HKEY_CLASSES_ROOT` is an alias. Four separate entries are created, although only the first of these applies to top-level OLEServers.

1. A sub-key named **dyalog.xxxx** where **xxxx** is the name of the OLEServer. This has a sub-key named **CLSID** whose *Default* value is a GUID corresponding to the ClassID property of the OLEServer.

2. A sub-key named **CLSID\xxxx** where **xxxx** is the GUID corresponding to the value of the ClassID property of the OLEServer. The *Default* value of this sub-key is the name of the OLEServer, and the sub-key itself contains sub-keys, namely **DyalogDispInterface**, **DyalogEventInterface, InProcHandler32**, **LocalServer32**, **ProgID**, **TypeLib**, and **VersionIndependentProgID**.

    2.1. **DyalogDispInterface** and **DyalogEventInterface** have their *Default* values set to the GUID for the Interface entry (see Paragraph 4). This GUID is generated internally by the registration of the Type Library.

    2.2. **InProcHandler32** has the *Default* value "OLE32.DLL".

    2.3. **LocalServer32** has its *Default* value set to the command line that is required to start the OLEServer. This is the full path-name of the appropriate DYALOG.EXE or DYALOGRT.EXE followed by the full path-name of the corresponding workspace plus any options that were specified in the *Create bound file* dialog box.

2.4. **ProgID** has its *Default* value set to "dyalog.xxxx" where "xxxx" is the name of the OLEServer.

2.5. **TypeLib** has its *Default* value set to the GUID corresponding to the TypeLibID property of the OLEServer.

2.6. **VersionIndependentProgID** has its *Default* value set to "dyalog.xxxx" where "xxxx" is the name of the OLEServer (same as **ProgID**).

2.7. Note that for a sub-object (an OLEServer that is a child of another OLEServer) only the **InProcHandler32** key is required, although the other entries are created and are in fact redundant.

3. A sub-key named **TypeLib\xxxx** where **xxxx** is the GUID corresponding to the value of the TypeLib property of the OLEServer. This contains a sub-key named **1.0** (which refers to its version number). The *Default* value of **1.0** is "Type Library for xxxx" where "xxxx" is the name of the OLEServer. **1.0** contains three further sub-keys named **0**, **FLAGS** and **HELPDIR**.

3.1. **0** (this identifies the language id; 0 refers to *all* languages) contains a sub-key named **win32** whose *Default* value is the full path-name of the Type Library file associated with the OLE object; i.e. the value of the TypeLibFile property of the OLEServer.

3.2. **FLAGS** has a *Default* value of "0".

3.3. **HELPDIR** has its *Default* value set to the full path-name of the directory in which the corresponding workspace is saved.

4. Sub-keys named **Interface\xxxx** where **xxxx** is the GUID referenced by the value of **DyalogDispInterface** and **DyalogEventInterface** described in paragraph 2. The *Default* values of these sub-keys is "xxxxdisp" where "xxxx" is the name of the OLEServer. You may identify the correct **Interface** sub-key by searching the registry for this string. It has three sub-keys named **ProxyStubClsid**, **ProxyStubClsid32**, and **TypeLib**.

4.1. **ProxyStubClsid** has a *Default* value of a GUID that references an interface of type PSDispatch.)

4.2. **ProxyStubClsid32** (same as **ProxyStubClsid**).

4.3. **TypeLib** has two values. Its *Default* value is the GUID identified by the TypeLib property of the OLEServer object, or, for a child OLEServer, the TypeLib property if its parent OLEServer. Its *Version* value is "1.0".

The LOAN Workspace

`LOAN.DWS` contains a single namespace called `Loan` which is used to calculate monthly repayments on a loan. As supplied, LOAN is a pure APL workspace. You will have to turn it into an OLE Server, and declare a method and a property, before you can use it.

The `Loan` namespace contains a single function `CalcPayments` and a variable `PeriodType`.

The `CalcPayments` function takes a 5-element numeric vector as an argument whose elements specify:

      1. loan amount

      2. maximum number of periods for repayment

      3. minimum number of periods for repayment

      4. maximum annual interest rate

      5. minimum annual interest rate

`CalcPayments` also uses the "global" variable `PeriodType` which specifies whether the periods (above) are years or months. This is done solely to illustrate how another application can manipulate an APL object via its variables (properties) as well as by calling its functions (methods).

`CalcPayments` returns a matrix. The first row contains the period numbers (from min to max). The first column contains the interest rates (from min to max in steps of 0.5%). Other elements contain the monthly repayments for the corresponding number of periods and interest rates.

Using CalcPayments

The following session transcript illustrates how the `CalcPayments` function is used.

```
      )LOAD LOAN
C:\Dyalog101\samples\ole\LOAN saved ...

      )OBS
  Loan
      )CS Loan
#.Loan
      )FNS
CalcPayments
      )VARS
PeriodType
```

```
      CalcPayments 10000 5 3 6 3
0     3           4           5
3   290.8120963 221.3432699 179.6869066
3.5 293.0207973 223.5600105 181.9174497
4   295.2398501 225.7905464 184.1652206
4.5 297.4692448 228.0348608 186.4301924
5   299.708971  230.2929357 188.7123364
5.5 301.959018  232.5647523 191.0116217
6   304.2193745 234.8502905 193.3280153
```

The CalcPayments Function

```
[0]    PAYMENTS←CalcPayments X;LoanAmt;LenMin;LenMax;IntrMin
                 ;IntrMax;PERIODS; INTEREST;NI;NM;PER;INT
[1]    ⍝ Calculates loan repayments
[2]    ⍝ Argument X specifies:
[3]        ⍝    LoanAmt      Loan amount
[4]        ⍝    LenMax       Maximum loan period
[5]        ⍝    LenMin       Minimum loan period
[6]        ⍝    IntrMax      Maximum interest rate
[7]        ⍝    IntrMin      Minimum interest rate
[8]     ⍝ Also uses the following global variable
[9]     ⍝   PeriodType  Type of period;1 = years, 2 = months
[10]
[11]   LoanAmt LenMax LenMin IntrMax IntrMin←X
[12]
[13]   PER←PERIODS←¯1+LenMin+⍳1+LenMax-LenMin
[14]   PERIODS←PERIODS×12 1[PeriodType]
[15]   INT←INTEREST←0.5×¯1+(2×IntrMin)+⍳1+2×IntrMax-IntrMin
[16]   INTEREST←INTEREST÷100×12 1[PeriodType]
[17]
[18]   NI←⍴INTEREST
[19]   NM←⍴PERIODS
[20]
[21]   PAYMENTS←(LoanAmt)×((NI,NM)⍴NM/INTEREST)÷
                         1-1÷(1+INTEREST)∘.*PERIODS
[22]   PAYMENTS←PER,[1]PAYMENTS
[23]   PAYMENTS←(0,INT),PAYMENTS
```

Registering Loan as an OLE Server

To use this example, you **must** first

1. Convert the `Loan` namespace into an OLEServer object.

2. Declare the COM properties for `CalcPayments` and `PeriodType`.

3. Create an in-process or out-of-process server and register the Loan object on your system.

Please perform the following steps:

`)LOAD` the `LOAN` workspace from the `samples\ole` sub-directory

```
      )LOAD SAMPLES\OLE\LOAN
samples\ole\loan saved ...
      )OBS
Loan
```

Execute the following statement to make `Loan` an OLEServer object:

```
      Loan.⎕WC 'OLEServer'
```

Now, using the COM Properties tab of the Properties dialog box, define the syntax and data types for the `CalcPayments` function and the `PeriodType` variable so that they are exported as a method and property respectively.
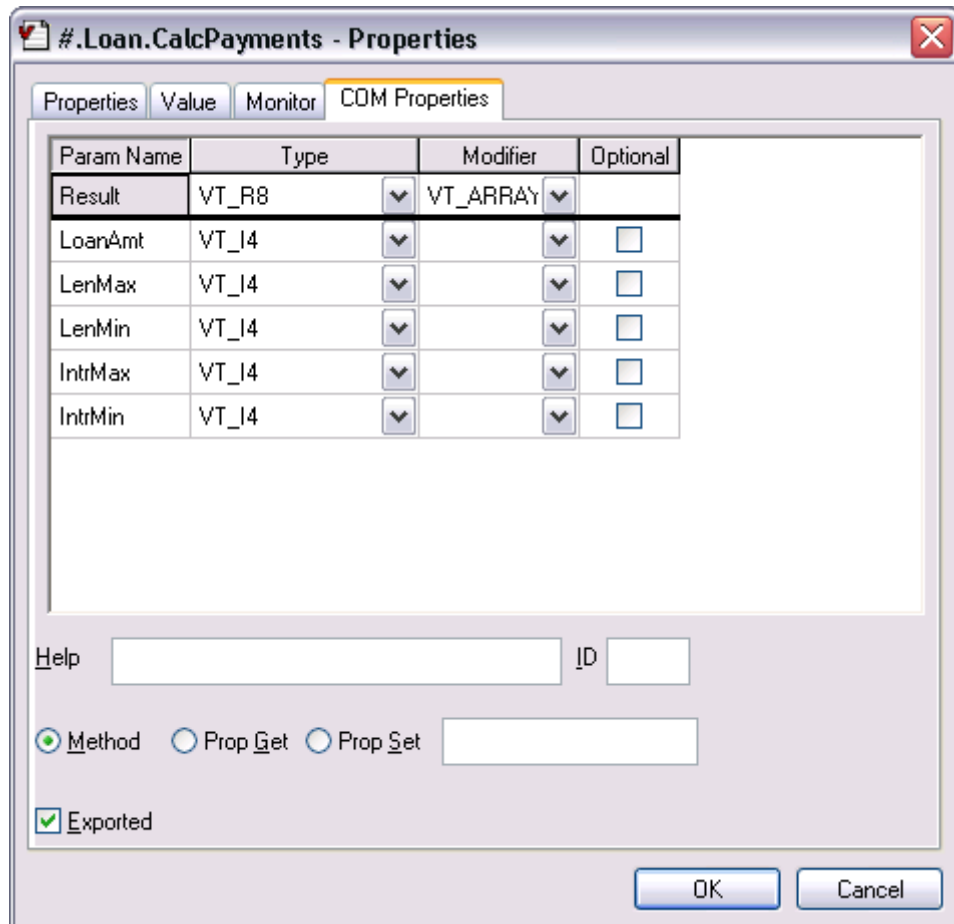
The picture above shows the COM properties that are required to export function `CalcPayments` as a method. The function is declared to require 5 parameters of type VT_I4 (integers) and return a result of type VT_ARRAY of VT_R8 (an array of floating-point numbers).

The names you choose for the parameters will be visible in an object browser and certain other programming environments.
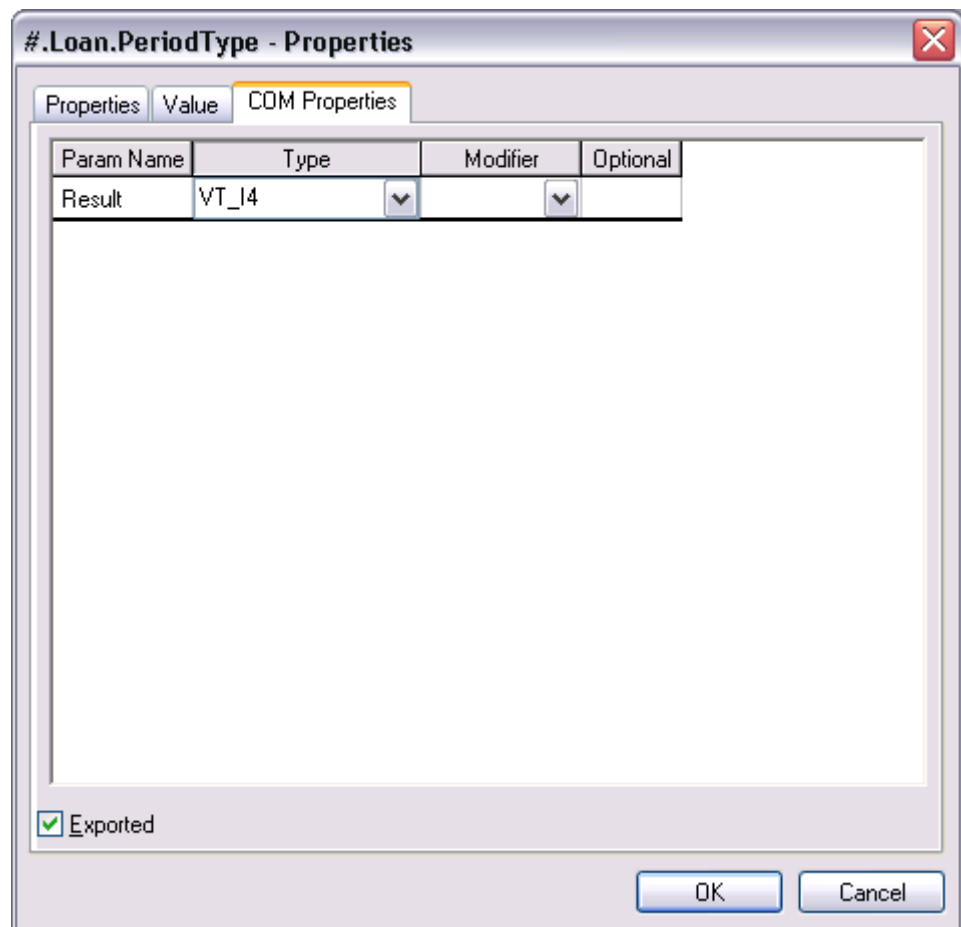
The picture above shows the COM properties to export variable `PeriodType` as a property. The property is declared to be of type VT_I4 (integer).

Rename and save the workspace to avoid overwriting the original:

```
      )WSID MYLOAN
was SAMPLES\OLE\LOAN
      )SAVE
MYLOAN saved ...
```

Finally, to create your OLE Server, choose *Export* from the Session *File* menu and complete the *Create bound file* dialog box as shown below. In this case, the OLE Server is created as an in-process server, bound to the development version of the Dyalog APL DLL (because the *Runtime application* checkbox is cleared)
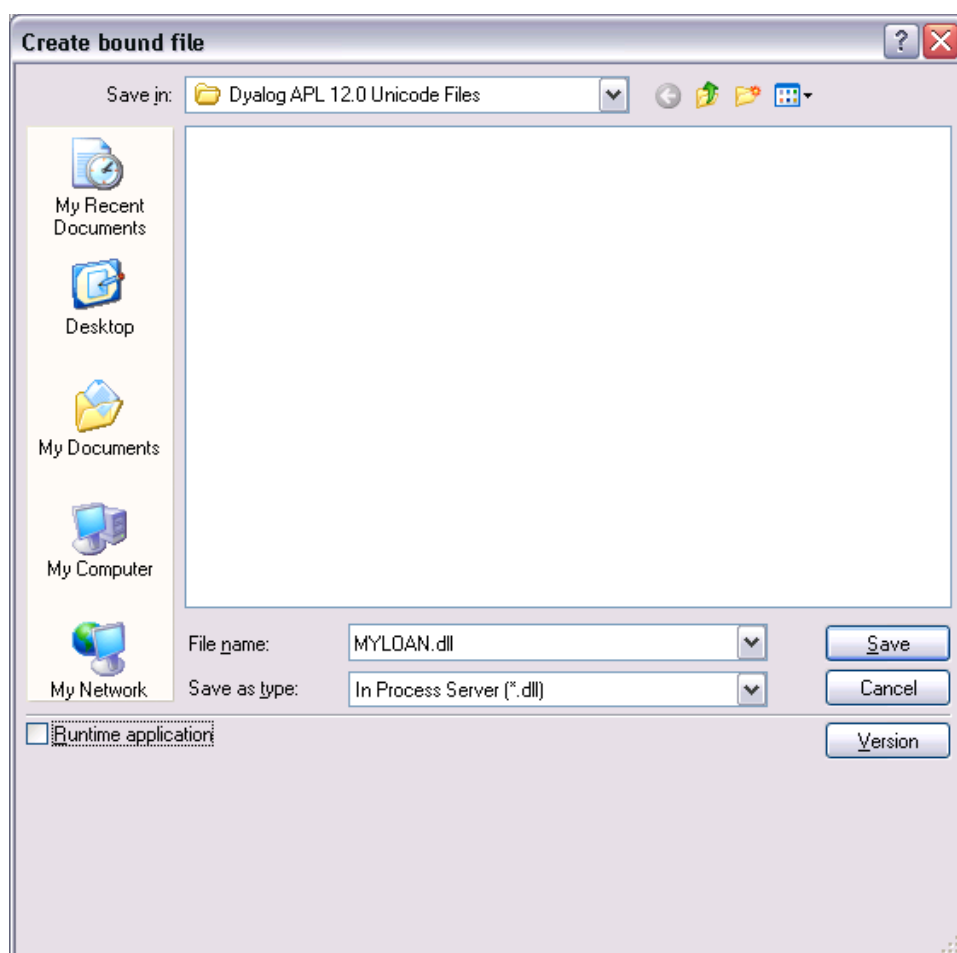
Note that appropriate information will be displayed in the Status window to inform you of the success (or failure) of the operation.

Using Loan from Visual Basic Version 4.0

Start Visual Basic Version 4.0 and open the project "aplLoan.vbp" that is provided in the Dyalog APL sub-directory `samples\ole\`.

This project is based upon an example that is distributed with Visual Basic (`vb\samples\grid\loan.vbp`) and is described in the Visual Basic Programmer's Guide Chapter 13.

To run the project, select Run/Start from the Visual Basic menu bar. Then enter data into the Visual Basic application as shown below. When you click *Show Payments*, Visual Basic actually calculates the repayments matrix by calling the CalcPayments method in the dyalog.Loan object; i.e. it runs the `CalcPayments` function in the `Loan` namespace.

How it Works

The original Visual Basic code has been modified so that instead of calculating loan repayments itself, it uses the Dyalog APL `Loan` object to perform this task. The changes required to the original Visual Basic code were as follows.

Firstly, the declarations section specifies a variable called APLLoan to be of type Object:

```
Dim APLLoan As Object
```

Secondly, the Load procedure for the Form (this module gets executed when the application starts) contains the statement:

```
Set APLLoan = CreateObject("dyalog.Loan")
```

The effect of these two statements is that when the application is run, VB asks OLE to provide the external object called *dyalog.Loan*. This name will have been recorded in the registry by Dyalog APL when you exported the `Loan` object. If you exported `Loan` as an *out-of-process* OLE Server, OLE starts the appropriate version (development or run-time) of Dyalog APL with your workspace `MYLOAN`. If you exported `Loan` as an *in-process* OLE Server, OLE loads MYLOAN.DLL into your Visual Basic application which in turn loads the appropriate Dyalog APL DLL. When it starts, Dyalog APL recognises that the request has come from OLE and responds by creating an instance of the `Loan` namespace which is connected to the Visual Basic process that has requested it.

The final change was to rewrite the VB subroutine that performed the calculations to call instead the Dyalog APL `Loan` object

The Revised CalcPmnts() Procedure

```
Private Sub CalcPmnts()
    Dim PeriodType
    ' Calculate number of periods and interest rates ..
    Periods = (LenMax - LenMin) + 1
    Rates = ((IntrMax - IntrMin) * 2) + 1
    If mnuOptLen(0).Checked = True Then
        PeriodType = 1
    Else
        PeriodType = 2
    End If
    ' Set PeriodType property of APLLoan object
    APLLoan.PeriodType = PeriodType
    ' Call Calc method of APLLoan object
    Payments = APLLoan.Calc(LoanAmt, LenMax, LenMin,
IntrMax, IntrMin)
End Sub
```

The first statement to note is:

APLLoan.PeriodType = 1

In Visual Basic terms, this statement sets the PeriodType property of the APLLoan object to the value of a local variable, also called PeriodType. What actually happens, is that the APL variable `PeriodType` in the corresponding running instance of the `Loan` namespace is set to the requested value.

The next statement:

Payments = APLLoan.CalcPayments(LoanAmt, LenMax, LenMin,
          IntrMax, IntrMin)

calls the APL function `CalcPayments` and receives the result.

In Visual Basic terms, this statement invokes the CalcPayments method of the APLLoan object. In practice, it calls the `CalcPayments` APL function with the specified argument and puts the result in the local variable Payments. Note that the conversion between the result of the function (a Dyalog APL floating-point matrix) and the corresponding Visual Basic data type is performed automatically for you.

Notice that the APLLoan variable is "global". This means that the dyalog.Loan object is loaded only once, when the Visual basic application is started and remains loaded until it is closed.

Using Loan from Excel

Start Excel and load the spreadsheet Loan.xls from the Dyalog APL sub-directory `samples\ole`.

The *Payments* button fires a simple macro that uses the APL dyalog.Loan object to perform repayment calculations.

This example is slightly simpler than the Visual Basic one in that the user does not specify a down payment, nor can the user change the period type, which is years.

To run the example enter data into the cells as shown below, then click *Payments*. When you do so, Excel runs the Calc macro and this causes OLE to initialise the dyalog.Loan OLE Server

The Calc macro actually calculates the repayments matrix by calling the CalcPayments method in the dyalog.Loan object; i.e. it runs the `CalcPayments` function in the `Loan` namespace.

loan.xls

	A	B	C	D	E	F	G
1	Loan Amount		120000		0	20	
2	Max Years in Loan		30		7	930.3587	910.1
3	Min Years in Loan		20		7.5	966.7118	946.9
4	Max Interest Rate		15		8	1003.728	984.5
5	Min Interest Rate		7		8.5	1041.388	1022
6					9	1079.671	1061
7			Payments		9.5	1118.557	1100
8					10	1158.026	1140
9					10.5	1198.056	1181

Sheet1 / Sheet2 / Sheet3 / Shee

How it Works

The Excel macro code is shown below.

```
Sub Calc()
    Dim APLLoan As Object
    Dim Payments As Variant
    Set APLLoan = CreateObject("dyalog.Loan")
    LoanAmt = Cells(1, 3).Value
    LenMax = Cells(2, 3).Value
    LenMin = Cells(3, 3).Value
    IntrMax = Cells(4, 3).Value
    IntrMin = Cells(5, 3).Value
    APLLoan.PeriodType = 1
    Payments = APLLoan.CalcPayments(LoanAmt, LenMax,
                        LenMin, IntrMax, IntrMin)
    For r = 0 To UBound(Payments, 1)
        For c = 0 To UBound(Payments, 2)
            Cells(r + 1, c + 5).Value = Payments(r, c)
        Next c
    Next r
End Sub
```

The statement:

Dim APLLoan As Object

declares a (local) variable called APLLoan to be of type Object

The next statement:

Set APLLoan = CreateObject("dyalog.Loan")

creates an instance of dyalog.Loan associated with this variable.

Effectively, when the macro is run, Excel asks OLE to provide the external object called *dyalog.Loan*.

 If you exported **Loan** as an *out-of-process* OLE Server, OLE starts the appropriate version (development or run-time) of Dyalog APL with your workspace MYLOAN. If you exported **Loan** as an *in-process* OLE Server, OLE loads MYLOAN.DLL into your Visual Basic application which in turn loads the appropriate Dyalog APL DLL. In either case, an instance of the **Loan** namespace is connected to the Excel macro as an Object.

The next statement to notice is:

```
APLLoan.PeriodType = 1
```

In Excel terms, this statement sets the PeriodType property of the APLLoan object to the value 1. What actually happens, is that the APL variable `PeriodType` in the corresponding running instance of the `Loan` namespace is set to 1.

Finally, the following statement:

```
Payments = APLLoan.CalcPayments(LoanAmt, LenMax, LenMin,
                                IntrMax, IntrMin)
```

calls the APL function `CalcPayments` and receives the result.

In Excel terms, this statement invokes the CalcPayments method of the APLLoan object. In practice, it calls the `CalcPayments` APL function with the specified argument and puts the result in the local variable Payments. Note that the conversion between the result of the function (a Dyalog APL floating-point matrix) and the corresponding Excel data type is performed automatically for you.

Notice that the APLLoan variable is local to the Calc macro. This means that the dyalog.Loan object is loaded every time that Calc is run and is unloaded when it terminates.

Using Loan from two applications

If you have exported `Loan` as an *out-of-process* OLE Server, you may use it concurrently from two or more client applications. To test this, follow the instructions for:

> 1) Using Loan from Visual Basic Version 4.0
>
> 2) Using Loan from Microsoft Excel

At this stage, you have two applications (Visual Basic and Excel) connected to a single Dyalog APL process with a single workspace `MYLOAN`.

Each application is actually connected to a separate running instance of the `Loan` namespace, although there is only one copy of the `CalcPayments` function in memory.

Only the variable `PeriodType` is duplicated in the two instances of `Loan`.

Note that the synchronisation of OLE requests from the different applications is entirely handled for you by Dyalog APL.

Notice too that this time there is no delay each time you run the Calc macro in Excel. When you are using Loan from Microsoft Excel on its own, there *is* a delay because OLE loads Dyalog APL and the `MYLOAN` workspace every time you run Calc This is because the APLLoan variable is local to Calc.

However, the corresponding Visual Basic variable is global and therefore forces OLE to keep dyalog.Loan loaded.

Using Loan from Dyalog APL

It is of course possible to use Dyalog APL as both an OLE Automation client and an OLE Automation Server.

To use the dyalog.Loan object, start Dyalog APL and then enter the following expressions in the Session window.

```
      'LN'⎕WC'OLEClient' 'dyalog.Loan'
      )OBS
LN
      )CS LN
#.LN
      )METHODS
CalcPayments
      )PROPS
PeriodType

      CalcPayments 10000 5 3 6 3
0     3           4           5
3     290.8120963 221.3432699 179.6869066
3.5   293.0207973 223.5600105 181.9174497
4     295.2398501 225.7905464 184.1652206
4.5   297.4692448 228.0348608 186.4301924
5     299.708971  230.2929357 188.7123364
5.5   301.959018  232.5647523 191.0116217
6     304.2193745 234.8502905 193.3280153
```

The statement:

```
      'LN'⎕WC'OLEClient' 'dyalog.Loan'
```

causes APL to ask OLE to provide the external object called *dyalog.Loan*. This name will have been recorded in the registry by Dyalog APL when you saved the MYLOAN workspace.

If you exported Loan as an *out-of-process* OLE Server, OLE starts a second Dyalog APL process (development or run-time) with your workspace MYLOAN. There are now two separate copies of Dyalog APL running; one is the client, the other the server.

If you exported Loan as an *in-process* OLE Server, OLE loads MYLOAN.DLL into the Dyalog APL program which in turn loads the appropriate Dyalog APL DLL. These DLLs are both are loaded into the same address space as the original APL process. In effect, you have two copies of APL (and two workspaces) running as a single program.

Note that in both cases, the mapping between the corresponding functions and variables is direct. Effectively, the client namespace LN is an instance of the server namespace Loan.

Implementing an Object Hierarchy

Despite the close correspondence between the object model and Dyalog APL namespace technology, there is one significant difference. OLE does not support object hierarchies in the sense that one object *contains* or *owns* another.

Instead you must implement object hierarchies using *properties* that refer to other objects and/or *methods* that return objects as results.

It is not possible to pass Dyalog APL namespace hierarchies through OLE because OLE does not support them. If you want to write an OLE Automation Server in APL that implements an object hierarchy, you must follow the OLE conventions for doing so.

You can pass an *instance* of a Dyalog APL OLEServer namespace to an OLE client as a ⎕OR, which can be the result of a function or the value of a variable. In order to be recognised as an OLE object, the namespace must be of type OLEServer.

In fact, when you export a workspace containing one or more OLEServer objects, any child OLEServer objects that they contain are registered too.

The CFILES Workspace (`samples\ole\cfiles.dws`) illustrates the use of an object hierarchy.

The CFILES Workspace

`CFILES.DWS` contains a single OLEServer namespace called `CFiles` which implements a basic object-oriented interface to Dyalog APL component files.

This example allows an OLE Client, such as Excel, to read and write APL component files. It is deliberately over-simplified but illustrates how an object hierarchy may be implemented.

Unlike the previous example, the CFILES workspace is supplied as a complete OLEServer with all of the COM properties for its methods already defined. All you have to do is to export it as a COM Server.

The `CFiles` namespace contains a single function `OpenFile` and a sub-namespace called `File` which is also an OLEServer. This namespace contains functions `FREAD`, `FREPLACE`, `FAPPEND` and `FSIZE`.

To use this Server, an OLE Client requests an instance of the `dyalog.CFiles` object.

To open a component file, an OLE Client calls OpenFile with the name of the file as its argument. This function opens the file and returns, not a file tie number as you might expect, but an instance of the File namespace which is associated with the file. As far as the client is concerned, this is a subsidiary OLE object of type `dyalog.File`.

To perform file operations, the client invokes the `FREAD`, `FREPLACE`, `FAPPEND` and `FSIZE` methods (functions) of the File object.

A more sophisticated example might expose each component as a subsidiary object too.

Registering CFiles as an OLE Server

In order to explore the use of an APL OLE Server using the CFILES workspace as an example, you must register the CFiles object on your system.

`)LOAD` the `CFILES` workspace from the `samples\ole` sub-directory

```
      )LOAD SAMPLES\OLE\CFILES
samples\ole\cfiles saved ...
      )OBS
CFiles
```

Then select *Export* from the Session *File* menu and create either an in-process or out-of process OLE Server.

The OpenFile Function

```
      ∇ FILE←OpenFile NAME;F;TIE
[1]    ⍝ Makes a new File object
[2]     TIE←1+⌈/0,⎕FNUMS
[3]     NAME ⎕FTIE TIE
[4]     File.TieNumber←TIE
[5]     File.Name←NAME
[6]     FILE←⎕OR'File'
      ∇
```

`OpenFile` takes the name of an existing component file and opens it exclusively using `⎕FTIE`.

It returns an instance of the `File` namespace that is associated with the file through the variable `TieNumber`. This is global to the `File` namespace.

`OpenFile[4]` sets the variable `TieNumber` in the `File` namespace to the tie number of the requested file.

`OpenFile[5]` sets the variable `Name` in the `File` namespace to the name of the requested file. This is not actually used.

`OpenFile[6]` creates an instance of the `File` namespace using `⎕OR` and returns it as the result.

Note that there is a separate instance of `File` for every file opened by every OLE Client that is connected. Each knows its own `TieNumber` and `Name`.

The COM Properties dialog box for `OpenFile` is shown below. The function is declared to take a single parameter called *FileName* whose data type is VT_BSTR (a string). The result of the function is of data type VT_DISPATCH. This data type is used to represent an object.
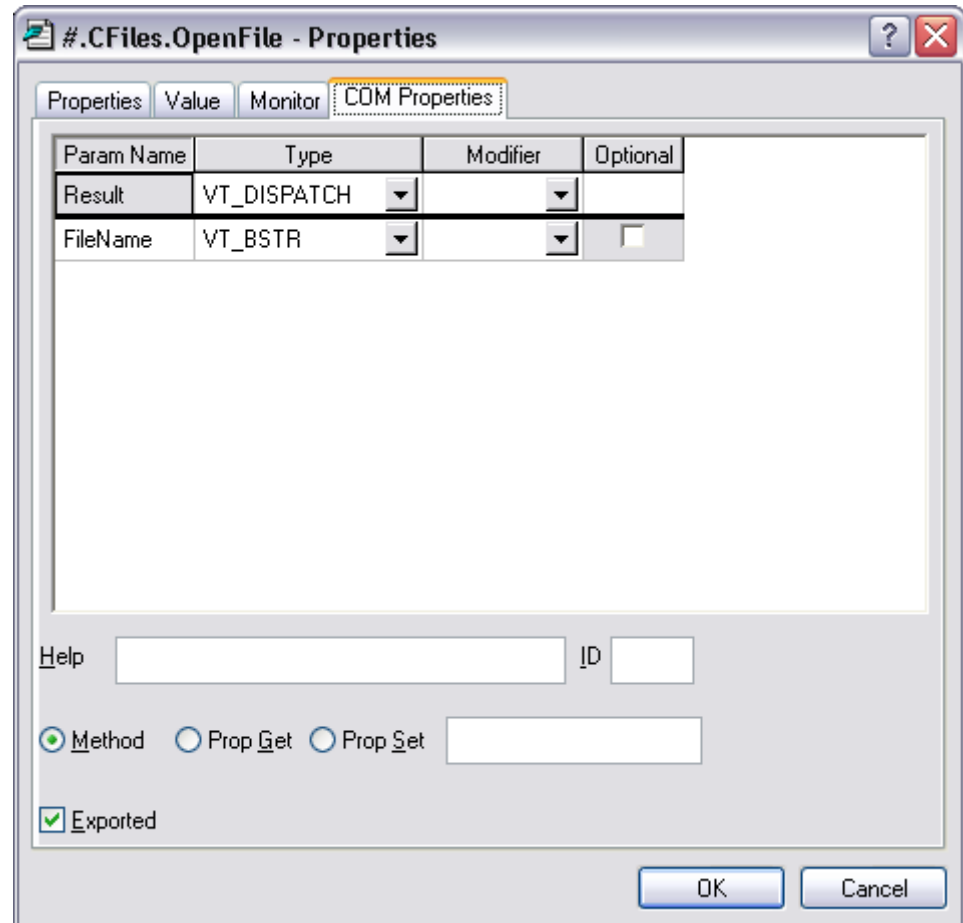
The FSIZE Function

```
      ∇ R←FSIZE
[1]     R←⎕FSIZE TieNumber
      ∇
```

`FSIZE` returns the result of `⎕FSIZE` for the file associated with the current instance of the File namespace.

The COM Properties dialog box for `FSIZE` is shown below. The function is declared to take no parameters. The result of the function is of data type VT_VARIANT. This data type is used to represent an arbitrary APL array.
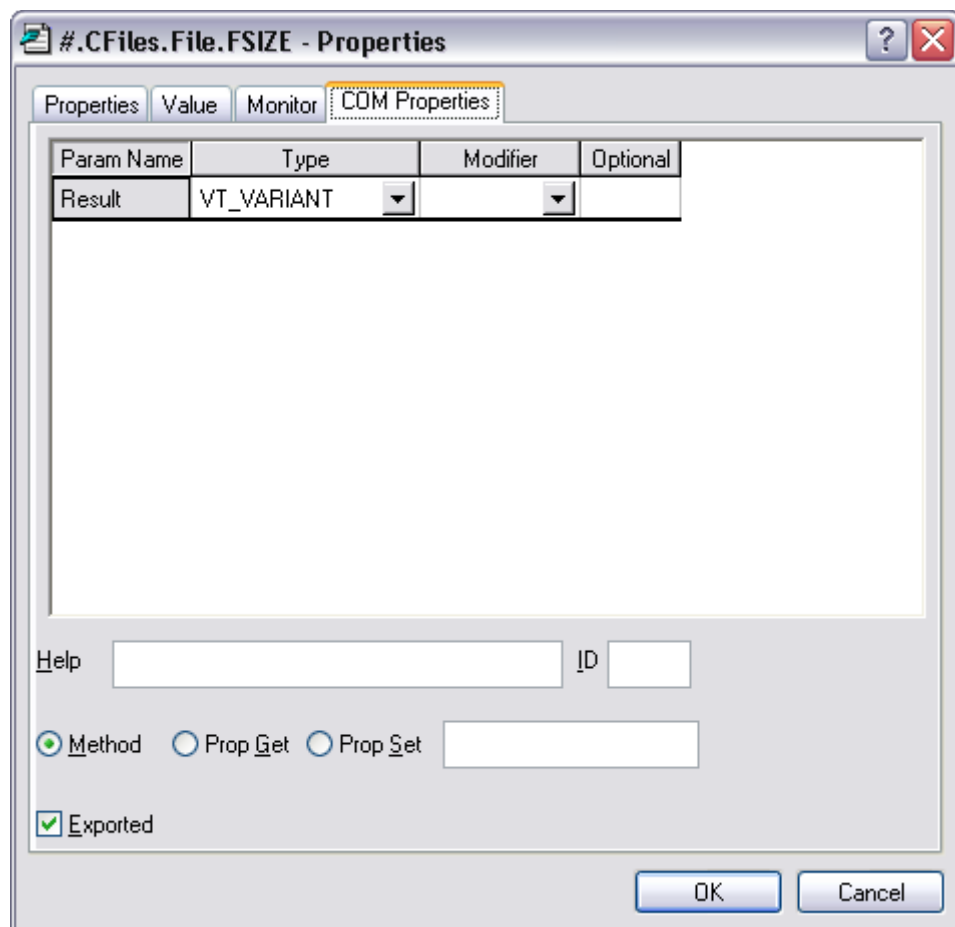
The FREAD Function

```
     ∇ R←FREAD N
[1]    R←⎕FREAD TieNumber,N
     ∇
```

FREAD returns the value in the specified component read from the file associated with the current instance of the File namespace.

The COM Properties dialog box for **FREAD** is shown below. The function is declared to take a single parameter called *Component* whose data type is VT_I4 (an integer). The result of the function is of data type VT_VARIANT. This data type is used to represent an arbitrary APL array.
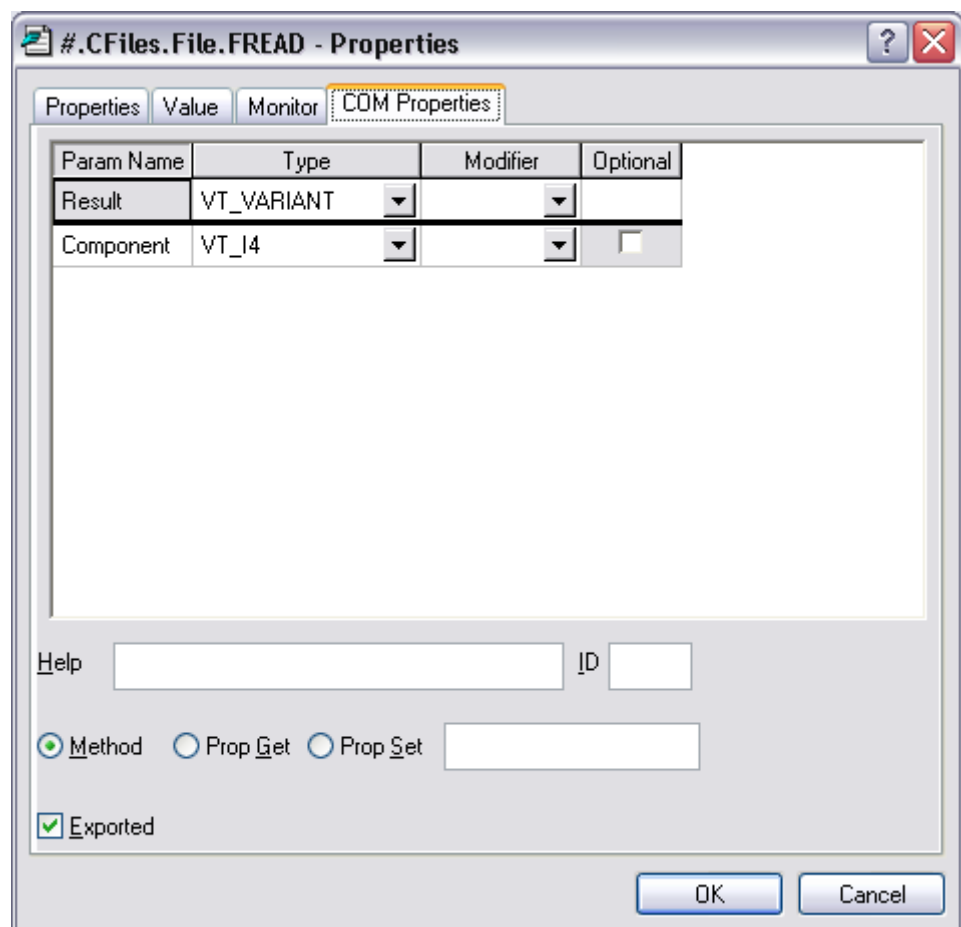
The FAPPEND Function

```
      ∇ R←FAPPEND DATA
[1]    R←DATA ⎕FAPPEND TieNumber
      ∇
```

`FAPPEND` appends a component onto of the file associated with the current instance of the File namespace.

The COM Properties dialog box for `FAPPEND` is shown below. The function is declared to take a single parameter called *Data* whose data type is VT_VARIANT. This data type is used to represent an arbitrary APL array. The result of the function is of data type VT_I4 (an integer).
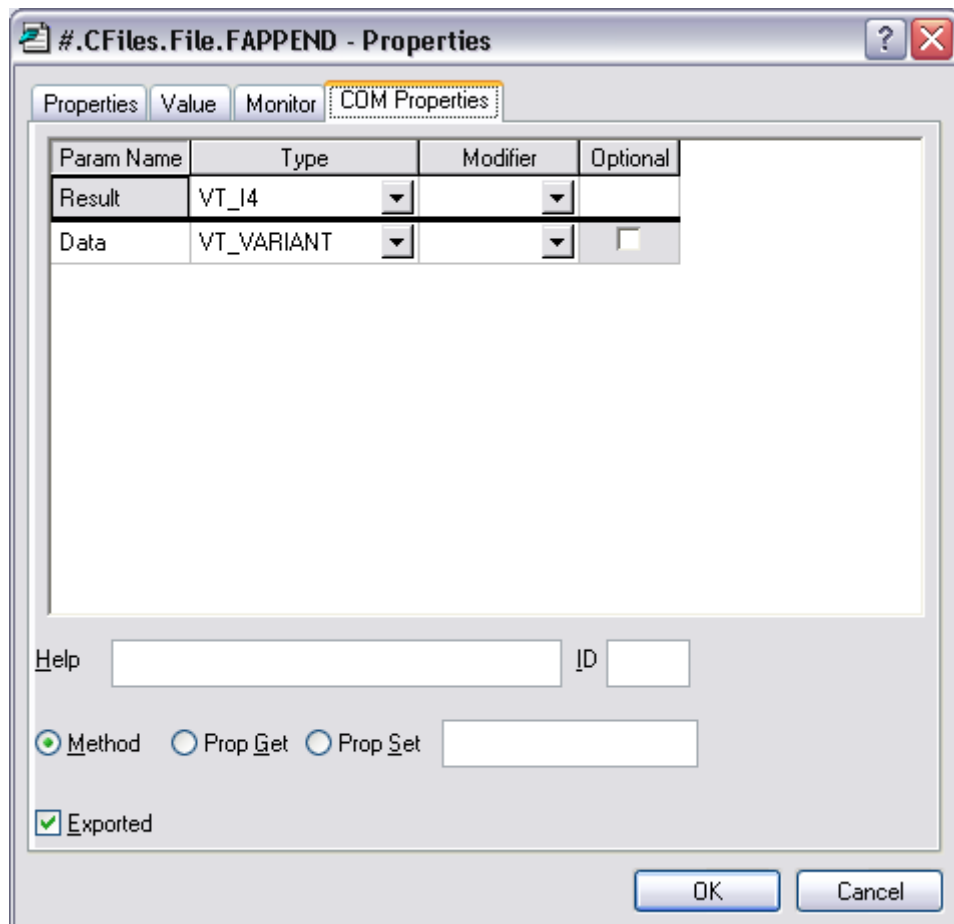
The FREPLACE Function

```
    ∇ FREPLACE ID;I;DATA
[1]    I DATA←ID
[2]    DATA ⎕FREPLACE TieNumber,I
    ∇
```

`FREPLACE` replaces the specified component of the file associated with the current instance of the File namespace.

The COM Properties dialog box for `FREPLACE` is shown below. The function is declared to take two parameter. The first is called *Component* and is of data type VT_I4 (integer). The second parameter is called *Data* and is of data type VT_VARIANT. This data type is used to represent an arbitrary APL array. The result of the function is of data type VT_VOID, which means that the function does not return a result.
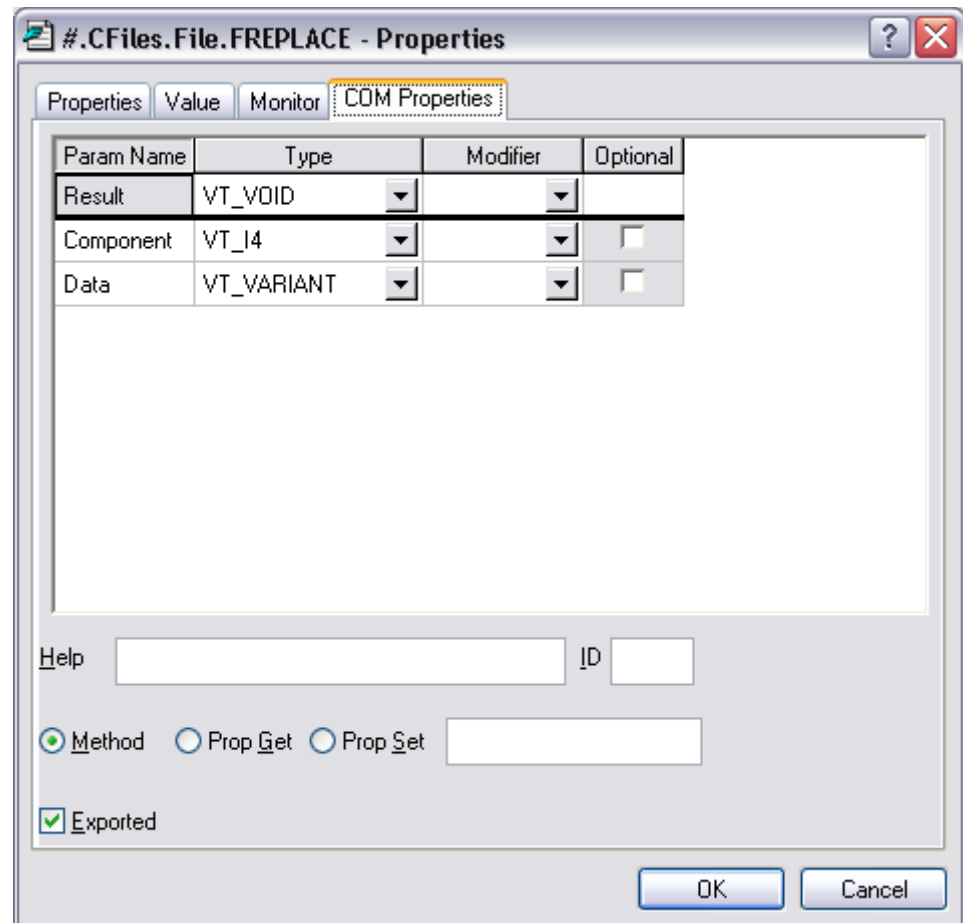
Using CFiles from Excel

Start Excel and load the spreadsheet CFiles.xls from the Dyalog APL sub-directory `samples\ole`.

Please note that to simplify the Excel code, only components containing matrices (such as those contained in samples\ole\test.dcf) are handled. Components containing scalars, vectors, higher-order arrays and complex nested arrays are not supported.

To open a file, type the name of the file and click the *Open* button. This runs the FOpen procedure. A test file named `test.dcf` is provided in the `samples\ole` sub-directory.

To read a component, enter the component number and click *Read*. This runs the FRead procedure.

To replace a component, first enter the component number. Then type some data elsewhere on the spreadsheet and select it. Now click *Replace*. This runs the FReplace procedure.

To append a component, enter some data elsewhere on the spreadsheet and select it. Now click *Append*. This runs the FAppend procedure.

The FOpen Procedure

```
Public CF As Object
Public File As Object

Dim Data As Variant

Sub FOpen()
    Set CF = CreateObject("dyalog.CFILES")
    f = Cells(1, 2).Value
    Set File = CF.OpenFile(f)
    Call FSize
End Sub
```

In the declaration section, the first statement declares a global variable CF to be of data type Object. This variable will be connected to the dyalog.CFiles OLE Server object. The second statement declares a global variable File to be of data type Object. This variable will be connected to the dyalog.File OLE Server object. The third statement declares a global variable Data to be of data type Variant. This is equivalent to a nested array. This variable will be used for the component data.

The statement:

```
        Set CF = CreateObject("dyalog.CFILES")
```

causes OLE to start Dyalog APL and obtain an instance of the dyalog.CFiles OLE Server object which is then associated with the variable CF. Because this variable is global, the OLE Server remains in memory and available for use.

The statement

```
        f = Cells(1, 2).Value
```

gets the name of the file to be opened and puts it into the (local) variable f.

Finally, the statement:

```
        Set File = CF.OpenFile(f)
```

calls the OpenFile function and stores the result (which is an object) in the global variable File.

The FRead Procedure

```
Sub FRead()
    c = Cells(4, 2).Value
    Data = File.FREAD(c)
    For r = 0 To UBound(Data, 1)
        For c = 0 To UBound(Data, 2)
            Cells(r + 6, c + 2).Value = Data(r, c)
        Next c
    Next r
End Sub
```

The statement:

```
    c = Cells(4, 2).Value
```

gets the number of the component to be read and stores it in the (local) variable c.

The statement:

```
    Data = File.FREAD(c)
```

This statement calls the `FREAD` function in the instance of the `File` namespace that is connected to the (global) Excel variable File. The result is stored in the variable Data.

The remaining statements copy the data from Data into the spreadsheet.

The FReplace Procedure

```
Sub FReplace()
    c = Cells(4, 2).Value
    Data = Selection.Value
    Call File.FReplace(c, Data)
    Call Fsize()
End Sub
```

The statement:

```
    c = Cells(4, 2).Value
```

gets the number of the component to be replaced and stores it in the (local) variable c.

The statement:

```
    Data = Selection.Value
```

gets the contents of the selected range of cells and stores it in the variable Data. This will be a 2-dimensional matrix.

The statement:

```
    Call File.FReplace(c, Data)
```

calls the FREPLACE function in the instance of the File namespace that is connected to the (global) Excel variable File.

The FAppend Procedure

```
Sub FAppend()
    Dim Rslt As Variant
    Data = Selection.Value
    Rslt = File.FAppend(Data)
    Call Fsize()
End Sub
```

The statement:

```
        Data = Selection.Value
```

gets the contents of the selected range of cells and stores it in the variable Data. This will be a 2-dimensional matrix.

The statement:

```
        Rslt = File.FAppend(Data)
```

calls the `FAPPEND` function in the instance of the `File` namespace that is connected to the (global) Excel variable File. The result of this function is ignored.

Configuring an *out-of-process* OLEServer for DCOM

Introduction

When you register an *out-of-process* OLEServer using *File/Export* or OLERegister, Dyalog APL automatically updates the Windows registry so that your OLEServer is immediately accessible to an OLE client application running on *the same* computer.

If you wish to make the same object accessible to client applications running on *different* computers (using distributed COM, or DCOM) you have to install additional registry entries on the server and on each of the clients.

Once you have established these registries entries, you should be able to access the OLEServer from Windows 95 or NT client computers in exactly the same way as if it were local; the client applications need not know where the server is located. In most cases, these additional registry entries are sufficient. However, the NT and DCOM security considerations may require the use of `dcomcnfg.exe` (a Microsoft utility) to set additional values. For example, if you get `E_ACCESSDENIED` errors when connecting from the client you may need to run `dcomcnfg.exe` on the server computer to assign the appropriate launch and access permissions for the OLEServer object.

The additional registry entries are described below. You may add these to the registry directly (using `regedit.exe`) or by running the functions provided in the `DCOMREG.DWS` workspace.

DCOM Registry Entries for the Server

On the computer upon which you want the OLEServer to be run, you must add the following registry entries.

1. A key under HKEY_CLASSES_ROOT\AppID whose name corresponds to the CLSID of your OLEServer object as reported by the value of its `ClassID` property. The *(Default)* value of this key should be the name of the server object. In addition, you must define a *RunAs* entry which specifies the manner in which a client application runs your server. The simplest choice is *Interactive User* which specifies that the client application is treated like a normal user.

    For example, if you had saved an OLEServer namespace called `Loan` (c.f. `samples\loan.dws`), whose `ClassID` property had the value {B80E9D40-2090-11D1-8F93-0020AFABD95D} the entries would be:

    ```
    HKEY_CLASSES_ROOT\AppID\{B80E9D40-2090-11D1-8F93-
    0020AFABD95D}
    (Default)=dyalog.Loan
    RunAs=Interactive User
    ```

2. An AppID entry to the HKEY_CLASSES_ROOT\CLSID key. (Note that this key will itself have been created by Dyalog APL/W when you saved the workspace) Once again, CLSID refers to the value of your OLEServer's `ClassID` property. The value of the AppID entry is the (same) CLSID.

Using the same example as above, the entry would be:

```
HKEY_CLASSES_ROOT\{B80E9D40-2090-11D1-8F93-
0020AFABD95D}
AppID={B80E9D40-2090-11D1-8F93-0020AFABD95D}
```

DCOM Registry Entries for the Client

On each of the computers from which you wish to call the OLEServer object as a client, you must add the following entries.

1. Two keys under HKEY_CLASSES_ROOT that identify the object (locally) and associate it with your OLEServer Note that the local name of the object is arbitrary and may be different on each client.

HKEY_CLASSES_ROOT\dyalog.ServerName
HKEY_CLASSES_ROOT\dyalog.ServerName\CLSID

CLSID is again the CLSID of the OLEServer object (this **must** be the same as that of the server machine). dyalog.ServerName can be replaced with whatever name you want clients to use to refer to this object

2. Under HKEY_CLASSES_ROOT\AppID, a key whose name corresponds to the CLSID of your server object. The *(Default)* value of this key should be the name of the OLE server object (its name on the server computer). In addition, the key should contain a *RemoteServerName* entry whose value is the name of the server computer. For example:

```
HKEY_CLASSES_ROOT\AppID\{B80E9D40-2090-11D1-8F93-
0020AFABD95D}
   (Default)=dyalog.Loan
   RemoteServerName=ntsvr
```

DCOMREG Workspace

The workspace DCOMREG.DWS contains a single namespace called `reg` that contains three functions to help register an *out-of-process* OLE Server for DCOM.

RegDCOMServer

This function should be run on the *server* computer and is called as follows:

```
RegDCOMServer ServerName CLSID
```

Where `ServerName` is a character string containing the (full) name of the OLEServer (e.g. `dyalog.Loan`) and `CLSID` is a character string containing the CLSID of the server (the value of it `ClassID` property). For example:

```
      )LOAD LOAN
.\LOAN saved ...
      )COPY DCOMREG
DCOMREG saved ...

      CLSID ←('Loan' ⎕WG 'ClassID')
      reg.RegDCOMServer 'dyalog.Loan' CLSID
```

RegDCOMClient

This function should be run on each of the *client* computers and is called as follows:

```
machine RegDCOMClient ServerName CLSID
```

Where `machine` is a character vector specifying the name of the (NT) server computer, `ServerName` is a character vector containing the (full) name of the OLEServer (e.g. `dyalog.Loan`) and `CLSID` is a character string containing the CLSID of the server (the value of it `ClassID` property). For example:

```
      CLSID←'{B80E9D40-2090-11D1-8F93-0020AFABD95D}'
      'NTSVR' reg.RegDCOMClient 'dyalog.Loan' CLSID
```

Config

This niladic function simply invokes the `dcomcnfg.exe` utility using `⎕CMD`.

Calling an OLE Function Asynchronously

Introduction

Functions exported by an OLEServer are executed (by the underlying OLE technology) in a *synchronous* manner. This means that the OLE client must wait for the function to complete before it can continue processing.

In certain cases the client may not be interested in a result from a function and it may be desirable for client not to have to wait. For example, if a function updates files or performs a printing task, it would be nice for the client application to continue while the server performs this task in background, or indeed (using DCOM) on another computer.

For an *out-of-process* OLE Server, this can be achieved by having the function that is called directly by the client *post* an event (using ☐NQ) onto the event queue and then return. When the function terminates, APL will take the next event from the queue and take the appropriate action. If the event has an associated callback function, APL will invoke it. Note that this happens immediately *after* the original function has terminated and a result (if any) has been returned to the client. This means that the APL OLEServer continues processing at the same time as the client application.

Note however that while the OLEServer is processing, further OLE requests will be queued. For example, if the client were to call the same function again immediately, the function would not be invoked until the original processing has finished and the client would therefore wait (note that OLE itself will actually time-out after a certain period). Nevertheless, this technique is an effective way to offload batch processing tasks to a second (background) APL process or to one running on a different computer.

The OLEASYNC Workspace

The OLEASYNC workspace illustrates this technique. It contains a single namespace called `Async` which exports 2 functions (methods), `PRINT` and `ASYNC` and two variables (properties) `ERRCODE` and `COPIES`.

The first function, `PRINT`, prints a specified number of test pages *in the background*. `PRINT` does not actually do any printing. All it does is to associate a second function `PRINT_CB` as a callback on a user-defined event 3001 (the choice of 3001 is purely arbitrary). It then posts an event 3001 onto the queue and returns 0 as its result.

The function also illustrates the use of :Trap. Should either of the statements on lines
[3] and [4] fail, the function terminates cleanly and returns ⎕DM instead.

```
      ∇ R←PRINT N
[1]    ⍝ Prints N test pages "in background"
[2]    :Trap 0
[3]        '.'⎕WS'Event' 3001 'PRINT_CB'N
[4]        ⎕NQ'.' 3001
[5]        (R ERRCODE)←0
[6]    :Else
[7]        (R ERRCODE)←⎕DM ⎕EN
[8]    :EndTrap
      ∇
```

The actual printing is performed by PRINT_CB *after* PRINT has returned to the client
and *while* the client itself continues processing. It too uses :Trap to terminate cleanly
should an error occur.

```
      ∇ N PRINT_CB MSG;PR;I;M
[1]    ⍝ Callback function : prints N test pages
[2]    :Trap 0
[3]        'PR'⎕WC'Printer'
[4]        :For I :In ⍳N
[5]            'PR.'⎕WC'Text'(20 60ρ'Testing')(0 0)
[6]            1 ⎕NQ'PR' 'NewPage'
[7]        :EndFor
[8]        ERRCODE←0
[9]    :Else
[10]        ERRCODE←⎕EN
[11]    :EndTrap
      ∇
```

Note that the client can (later) query ERRCODE to find out whether or not the operation
succeeded. Indeed, referencing ERRCODE will synchronise the client and server
because the server will have to wait until PRINT_CB completes before it can service
the request for the value of ERRCODE.

The ASYNC function illustrates a slightly different approach and may be used to execute
any expression asynchronously. It simply associates its argument (a character vector) as
an expression to be executed when (user-defined) event 3001 occurs. It then *posts* this
event onto the queue as before.

```
      ∇ R←ASYNC CMD
[1]    ⍝ Executes expression CMD "asynchronously"
[2]    :Trap 0
[3]        '#.Async'⎕WS'Event' 3001 'DO'CMD
[4]        ⎕NQ'#.Async' 3001
[5]        (R ERRCODE)←0
[6]    :Else
[7]        (R ERRCODE)←⎕DM ⎕EN
[8]    :EndTrap
      ∇
```

The callback function `DO` is invoked (later) when the event 3001 is processed from the event queue. This happens immediately after the function `ASYNC` has returned its result to the client workspace. `DO` simply executes its left argument, which is the string that was supplied as the right argument to `ASYNC`.

```
      ∇ CMD DO MSG
[1]    :Trap 0
[2]        ⍎CMD
[3]        ERRCODE←0
[4]    :Else
[5]        ERRCODE←⎕EN
[6]    :EndTrap
    ∇
```

You may wonder why it is necessary to use a callback function as opposed to an execute expression. In particular, why not have `ASYNC[3]` as follows?

```
[3]           '#.Async'⎕WS'Event' 3001 '⍎CMD'
```

The reason is that whilst a callback will execute in the *instance* of the OLEServer namespace connected to this client (which is what we want), an execute expression will be executed in the master OLEServer namespace itself.

The namespace contains a fourth function called `LPR` which is designed to be called via `ASYNC` using an expression such as `ASYNC 'LPR'`.

```
      ∇ LPR;PR;I
[1]    :Trap 0
[2]        'PR'⎕WC'Printer'
[3]        :For I :In ιCOPIES
[4]            'PR.'⎕WC'Text'(20 60ρ'Testing')(0 0)
[5]            1 ⎕NQ'PR' 'NewPage'
[6]        :EndFor
[7]        ERRCODE←0
[8]    :Else
[9]        ERRCODE←⎕EN
[10]   :EndTrap
    ∇
```

Note that the number of copies to be printed is defined by the (global) variable `COPIES` whose default value is 1. This is done only to illustrate that `LPR` called via `DO` runs in the correct instance of the OLEServer (using *your* value of `COPIES`) as opposed to in the master OLEServer namespace itself.

Testing dyalog.Async

Load OLEASYNC and then register dyalog.Async as an OLE object by doing the following:

```
Async.⎕WC 'OLEServer'
'Async'⎕WS'ExportedFns' ('PRINT' 'ASYNC')
'Async'⎕WS'ExportedVars' ('ERRCODE' 'COPIES')
```

Rename the workspace to avoid overwriting the original and *)SAVE* it.

```
)WSID MYASYNC
)SAVE
```

Finally, register the OLE Server using *File/Export*. **Note that dyalog.Async will only work as an out-of-process OLE Server.**

Now clear the workspace and test dyalog.Async using Dyalog APL as an OLE client application. You could also try calling it from Excel. Note that the results from the functions PRINT and ASYNC are returned immediately.

```
      )CLEAR
clear ws
      'TEST' ⎕WC 'OLEClient' 'dyalog.Async'

      TEST.PRINT 3
0
      TEST.ERRCODE
0
      TEST.COPIES←2
      TEST.COPIES
2
      TEST.ASYNC 'LPR'
0
      TEST.ASYNC 99 ⍝ Wrong !
0
      TEST.ERRCODE
11
```

CHAPTER 13

Writing ActiveX Controls in Dyalog APL

An A*ctiveX Control* is basically a user-defined control that may be included in GUI applications and Web Browsers.

This chapter describes how you write ActiveX Controls in Dyalog APL/W.

A Dyalog APL ActiveX Control can be used by any other application that supports ActiveX. Such applications include Microsoft Visual Basic, Microsoft Excel, Microsoft Internet Explorer, Netscape Navigator with the NCompass ScriptActive Plug-In and of course Dyalog APL itself.

This chapter also includes a tutorial which teaches you how to:

- Create an ActiveX control in Dyalog APL
- Define and Export Properties, Methods and Events
- Include your ActiveX control in a Visual basic application
- Run your ActiveX control from a Web Browser.

Please note that the Visual Basic examples described in this Chapter were developed and tested with Visual Basic Version 5 and may require some modification to work with other versions of Visual basic.

Overview

What is an ActiveX Control ?

An ActiveX Control is a dynamic link library that represents a particular type of COM object. When an ActiveX Control is loaded by a host application, it runs *in-process*, i.e. it is part of the host application's address space. Furthermore, an ActiveX Control typically has a *window* associated with it, which normally appears on the screen and has a user interface.

An ActiveX Control is usually stored in file with the extension .OCX. The functionality provided by the control can be supplied entirely by functions in that file alone, or can be provided by other dynamic link libraries that it loads and calls, i.e. an ActiveX Control can be stand-alone or can rely on one or more other dynamic link libraries.

What is a Dyalog APL ActiveX Control ?

An ActiveX Control written using Dyalog APL is also a file with a .OCX extension. The file combines a small dynamic link library *stub* and a workspace. The functionality of the control is provided by the functions and variables in the workspace combined with a dynamic link library version of Dyalog APL named `DYALOG120.DLL` or `DYALOG120RT.DLL` that is normally installed in the Windows System directory.

Note that an ActiveX Control written in Dyalog APL is a GUI object that has a visible appearance and a user interface.

To write an ActiveX Control in Dyalog APL, you use `⎕WC` to create an ActiveXControl object, as a child of a Form. An ActiveXControl is a container object, akin to a Group or a SubForm, that may contain a whole range of other controls such as Edit, Combo, Button and Grid objects. You may populate your ActiveXControl with other objects at this stage and save them in the workspace. However, you may prefer to create these sub-objects when an instance of the ActiveXControl is created. This happens when your control is loaded by a host application.

All the functions and variables that represent methods and properties through which the ActiveXControl object exports its functionality, reside within the ActiveXControl namespace.

You may turn a workspace containing one or more ActiveXControl objects into an installable OCX file by selecting the *Export* menu item from the Session *File* menu.

Note that a single OCX file can therefore contain a number of ActiveX Controls.

When a Dyalog APL ActiveX Control is loaded by a host application, functions in the stub load the appropriate Dyalog APL dynamic link library into the host application. This in turn copies the appropriate parts of the workspace from the .OCX. If the same host application starts a second (different) ActiveX Control written in Dyalog APL, the appropriate parts of the second workspace are merged with the first. For further details, see the section entitled Workspace Management.

The Dyalog APL DLL

ActiveXControls are hosted (executed) by the Dyalog APL DLL. For further details, see *User Guide, Chapter 2*.

Instance Creation

When a host application creates an instance of an ActiveXControl object, the new instance generates a Create event. It is recommended that you make any GUI objects that you need within the ActiveXControl *at this stage*, rather than making them in advance and saving them in the workspace.

The reason for this is that until an instance of an ActiveXControl is created, its Size and *ambient properties* are not known. Ambient properties include the font (which may affect the size and position of the internal controls) and background and foreground colours. These are specified by the host application and should normally be honoured by the ActiveXControl. Although when you are developing an ActiveXControl it will have a specific size, the size of an *instance* of the object cannot be predicted in advance because it is determined by the host application. This alone is sufficient reason to delay the creation of sub-objects inside the ActiveXControl until the Create event occurs.

In addition to the Create event, ActiveXControl objects support a PreCreate event. This event is always generated before a Create event and signals the creation of a newly cloned namespace. However, it is reported *before* the host application has assigned it a *window*. You may therefore not use the PreCreate event to create sub-objects, but you may use it for other initialisation tasks if applicable. Many host applications distinguish between design mode, when the user may just place controls in a GUI framework, and run mode when the controls become fully active. Some applications, such as Microsoft Access, do not require the control to appear fully in design mode, but instead represent the control by a simple rectangle or bitmap. In these cases, the ActiveXControl will generate a PreCreate event in design mode and not generate a Create event until run-time. Others, like Visual Basic, require that the control appears in design mode as it would appear in the final application. In these cases, the Create event follows immediately after PreCreate.

Properties, Methods and Events

Typically, an ActiveX Control provides Properties, Methods and Events that allow the control to be configured and controlled by a host application.

The information about the properties, methods and events exported by an ActiveX Control is normally stored in its .OCX file. The information includes the name of each property and its data type, and the name and data type of each method and each of its arguments. The information for an event is similar to that for a method. In addition to these obligatory items, it is possible to include help strings and help ids which provide on-line documentation for the host application programmer

Dyalog APL provides facilities for you to specify all this information in one of two ways; using dialog boxes or by calling methods.

Firstly, the Properties dialog box for an ActiveXControl object includes three additional tabs named *COM Properties*, *COM Functions* and *COM Events*. These dialog boxes allow you to export variables as properties, to export functions as either properties or methods, and to export events. In addition, the individual Properties dialog boxes for all the functions and variables in an ActiveXControl namespace have an additional *COM Properties* tab which performs the same function. Examples of these dialog boxes are provided in the tutorial section of this chapter.

Secondly, the ActiveXControl object provides three (internal) methods that allow you to specify this information by executing APL statements. These methods are named SetVarInfo, SetFnInfo and SetEventInfo and examples of their use is given in the tutorial.

Generating Events

Events that are generated by Dyalog APL GUI objects *inside* an ActiveXControl are purely internal events and are not detectable by a host application. However, an ActiveXControl object may generate an arbitrary event for a host application using ⎕NQ with a left argument of 4.

An external event must have a name (numbers are not allowed) and may include one or more parameters that supply additional information. The name of the event and the name and data types of each of its parameters must be defined in advance using the COM Events tab of the Properties dialog box of the ActiveXControl object, or by calling its SetEventInfo method.

For example, the Dual control described in the tutorial has an event called ChangeValue1. This event supplies a parameter named Value1 that has a data type of VT_PTR to VT_I4 (pointer to an integer). The Dual control *generates* the event for the host application by executing the statement:

```
    4 ⎕NQ '' 'ChangeValue1' Value1
```

where `Value1` is the new value of its internal Slider control.

A host application may choose to ignore an event generated by an ActiveXControl, or it may attach a callback function that performs some action in response to the event. A callback function in the host application receives the parameters supplied by the event as parameters to the function. If the host application is Dyalog APL itself, the callback function receives the parameters as part of the event message.

A host application callback function may not return a result. However, it may modify any of the parameters that were supplied as part of the event message if those parameters are defined as pointers (VT_PTR to xxx).

The result of `4 ⎕NQ` is therefore a vector whose elements correspond to the pointer parameters in the order they were specified. The result does not contain elements for parameters that were not exported as pointers and may therefore be empty. In the above example, the result of `4 ⎕NQ` is a 1-element vector containing the, possibly modified, value of `Value1`.

The Dual Control Tutorial

The ActiveX control we will use in this example is deliberately an extremely simple one; so that the intricacies of the control itself do not get in the way of the principles involved. In practice, there are actually very few restrictions concerning the complexity of the ActiveX control, and it is perfectly possible to package complete multiple-window Dyalog APL applications in this way.

Your ActiveX control will be called a *Dyalog Dual Control* and is based on the Dyalog APL TrackBar object.

The Dual control allows the user to enter a number using a slider, whilst displaying its value in two different units. For example, you could use it to enter a temperature value which is displayed in both Centigrade and Fahrenheit units. Equally, the same control could be used to enter a measurement of length which is concurrently displayed in centimetres and inches.
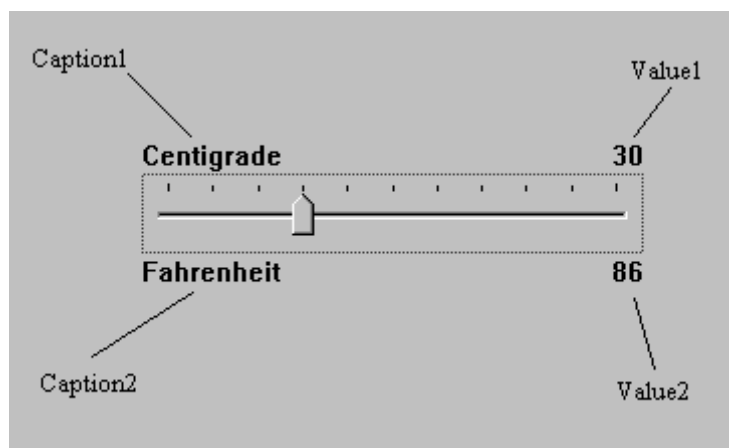
Methods

None. (The Dual control provides no methods.)

Properties

The Dual control provides the following properties:

Property	Description
Caption1	A text string that describes the primary units. This is displayed in the top left corner of the object.
Caption 2	A text string that describes the secondary or derived units. This is displayed in the bottom left corner of the object.
Value1	The current value of the control measured in primary units
Value2	The current value of the control measured in secondary units.
Intercept	Used to derive Value2 from Value1
Gradient	Used to derive Value2 from Value1
Min	The minimum value of Value1
Max	The maximum value of Value1

Value2 is derived from Value 1 using the expression:

```
Value2←Intercept+Gradient×Value1
```

Events

Your Dual control will generate a *ChangeValue1* event whenever the user alters Value1 using the slider.

The event message will contain a single parameter (the new value) which may be modified by the host application.

In other words, every time the value in the control changes, the host application may detect this as an event and has the opportunity to override the user.

Your Dual control will also generate a *ChangeValue2* event whenever the derived value in the control (Value2) changes. This event is reported for information only.

Introducing the Dual Control

To save time, the basic APL code for the Dual control has already been written. However, you will have to turn it into an ActiveX control yourself.

1. Load the DUALBASE workspace:

    ```
    )LOAD SAMPLES\ACTIVEX\DUALBASE
    ```

2. Run the function TEST and observe how the 2 Dual controls behave.

3. View the function `TEST` and observe how 2 separate instances of the `Dual` namespace `F.D1` and `F.D2` have been created using `⎕OR` and `⎕NS`.

4. Using the Dyalog APL Workspace Explorer, open up the various namespaces. See how `F.D1` and `F.D2` are clones of `Dual`.

5. Open the function `Dual.Create` and see how the individual components of the control are defined.

6. Close the Form `F`.

Changing Dual into an ActiveX Control

1. Change the name of the workspace to DUAL:

    ```
    )WSID DUAL
    ```

2. Make a new namespace called `F`

    ```
    )NS F
    ```

3. Using the Workspace Explorer, move the `Dual` namespace into `F`, so that `Dual` is a child namespace of `F`.

4. Now edit the function `F.Dual.Create` and make the following changes:

5. Remove all references to the local variable `POSITION`. This change is required because an ActiveX control has no say in its position within its parent. (Hint: use the Search/Replace dialog to remove all occurrences of `POSITION+`)

6. Remove the right argument, `SIZE and` change `Create[1]` from:

    ```
    H W←SIZE
    ```
    to
    ```
    H W←Size
    ```
    This change allows the control to fit itself within the space allocated by the host application.

7. Change Create[4] from:

    ```
    CH←⊃##.GetTextSize 'W'
    ```
    to
    ```
    CH←⊃GetTextSize 'W'
    ```

    The original code was designed to pick up the character height from the parent Form. The ActiveXControl object does this automatically via its own GetTextSize method.

    After making these changes, the Create function should be as follows:

    ```
         ∇ Create;H;W;POS;SH;CH;Y1;Y2
    [1]    H W←Size
    [2]    SH←40 ⍝ Default Trackbar height
    [3]    POS←2↑⌊0.5×0⌈(H-SH)
    [4]    CH←⊃GetTextSize'W'
    ```

```
[5]     'Slider'⎕WC'TrackBar'(POS)('Size'SH W)
[6]     Slider.(Limits AutoConf)←(Min,Max)0
[7]     Slider.(TickSpacing TickAlign)←10 'Top'
[8]     Slider.onThumbDrag←'ChangeValue'
[9]     Slider.onScroll←'ChangeValue'
[10]    Y1←POS[1]-CH+1
[11]    Y2←POS[1]+SH+1
[12]    'Cap1'⎕WC'Text'Caption1(Y1,0)('AutoConf' 0)
[13]    'Cap2'⎕WC'Text'Caption2(Y2,0)('AutoConf' 0)
[14]    'V1'⎕WC'Text'(⍕Value1)(Y1,W)
                    ('HAlign' 2)('AutoConf' 0)
[15]    CalcValue2
[16]    'V2'⎕WC'Text'(⍕Value2)(Y2,W)
                    ('HAlign' 2)('AutoConf' 0)
        ∇
```

8. Open the function F.Dual.Build. This function turns the Dual's parent namespace into a Form (an ActiveXControl currently requires a parent Form) and turns Dual itself into an ActiveXControl. It then attaches functions Create and Configure as callbacks on the Create and Configure events of the ActiveXControl object itself.

```
    ∇ Build
[1]    ##.⎕WC'Form'('Coord' 'Pixel')('KeepOnClose' 1)
[2]    ⎕WC'ActiveXControl'('Size' 80 200)
                        ('KeepOnClose' 1)
[3]    ⎕WS'Event' 'Create' 'Create'
[4]    ⎕WS'Event' 'Configure' 'Configure'
[5]    ⎕NQ'' 'Create'
        ∇
```

9. Run function `F.Dual.Build`. You should see a Form containing a single instance of the Dual control. Please resist any temptation to play with it at this stage; we want it to be in its default state for when we save it.

10. Type the following expression; note that the ClassID, which uniquely identifies your control, is allocated when you create the ActiveXControl object.

    `F.Dual.ClassID`

11. Save the workspace (DUAL.DWS).

12. From the Session File menu, select Export, choose where you want to save your OCX, and then click Save. It is a good idea to clear the Runtime application checkbox so that you can debug the control if anything goes wrong.
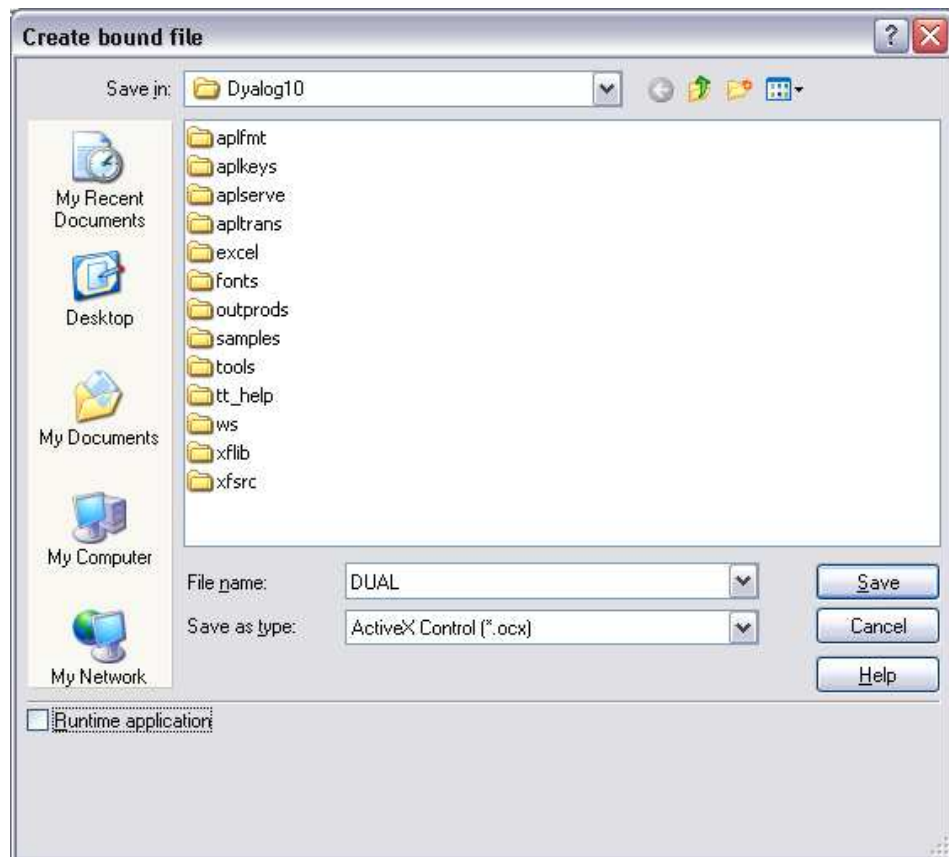
Testing the Dual Control

1. Close Dyalog APL

2. Start Visual Basic and click Standard EXE.

3. Click the right mouse button in the leftmost toolbox and select *Components* from the pop-up menu.

4. Locate the control named *Dyalog DUAL Objects*, set its check box on and click OK. This adds a tool for the Dual control to the Visual Basic designer toolbox.

5. Click on the new tool and drag out an outline on your Form. An instance of the Dual control will appear.

6. Repeat step 5 to position a second instance of the Dual control on your Visual Basic Form.

7. Select Run/Start from the Visual Basic menubar.

8. Exercise the 2 Dual controls.

9. Select Run/End from the Visual Basic menubar.

10. Click on one of the Dual controls and scroll through its Property list. Notice that all the properties listed are standard Visual Basic ones; there are no properties (or indeed methods and events) exported. We will learn how to do this next.

11. Close Visual Basic; do **not** save the project.

Defining and Exporting Properties

1. Start Dyalog APL and load the DUAL workspace (Hint: use the File menu; it will be the most recently saved file).

2. Change space into the `F.Dual` namespace.

    ```
    )CS F.Dual
    ```

3. The properties we wish to export are:

Caption1	Description of the primary set of units
Caption2	Description of the secondary set of units
Value1	The primary value in the control
Min	Minimum for Value1
Max	Maximum for Value1
Intercept	Used to derive the secondary value (Value2)
Gradient	Used to derive the secondary value (Value2)

Although we could export all these properties as *variables*, it is generally more useful to employ *Get* and *Put* functions. The reason for this is that there is no mechanism to detect when the host application changes a property/variable; nor is there any mechanism to prevent it assigning an inappropriate value. The *Get* and *Put* functions you need are listed below. To save you time, you can copy them in from the workspace DUALFNS.

```
     )COPY SAMPLES\ACTIVEX\DUALFNS

        ∇ R←GetCaption1
[1]     R←Caption1

        ∇ SetCaption1 C
[1]     Cap1.Text←Caption1←C

        ∇ R←GetCaption2
[1]     R←Caption2

        ∇ SetCaption2 C
[1]     Cap2.Text←Caption2←C

        ∇ R←GetIntercept
[1]     R←Intercept

        ∇ SetIntercept I
[1]     Intercept←I
[2]     CalcValue2
[3]     V2.Text←⍕Value2

        ∇ R←GetGradient
[1]     R←Gradient

        ∇ SetGradient G
[1]     Gradient←G
[2]     CalcValue2
[3]     V2.Text←⍕Value2

        ∇ R←GetValue1
[1]     R←Value1

        ∇ SetValue1 V
[1]     1 ⎕NQ'' 'ChangeValue'V
```

The *Get* functions need no explanation; they simply return the value of the corresponding variable. The *Set* functions assign a new value to the corresponding variable **and** update the control accordingly. `SetValue1` does this by enqueuing a Scroll event to the Slider, which in turn invokes the `ChangeValue` callback.

4. Display the Object Properties dialog box for the function `GetCaption1`. (Hint: use the Workspace Explorer).

5. Select the *COM Properties* tab. As you have not yet defined any OLE attributes, the default display is as follows:
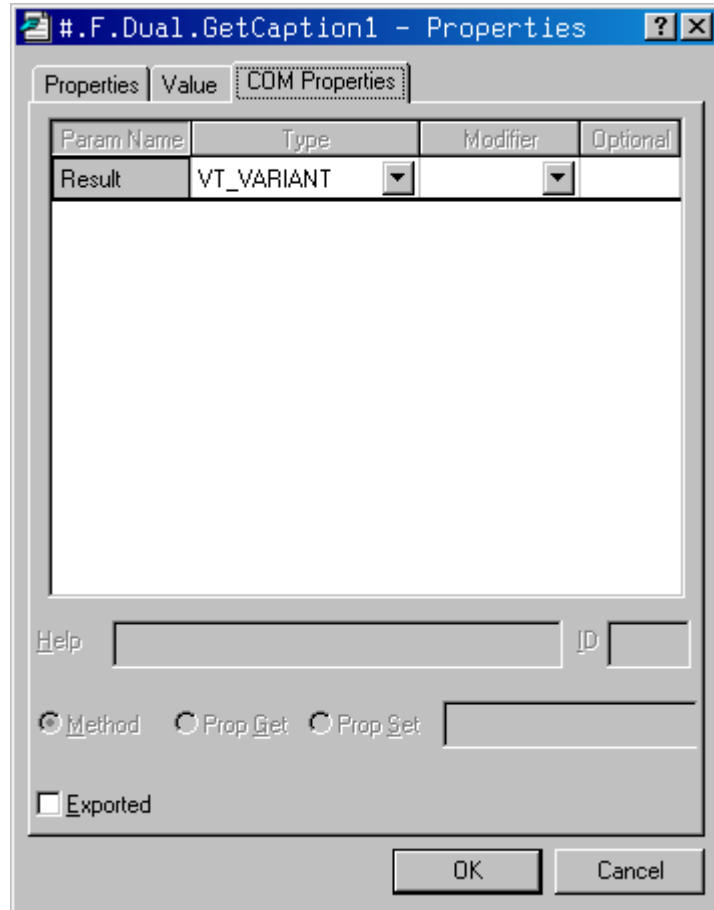
Check the *Exported* option button.

Change the data type for the *Result* to *VT_BSTR* (a text string).

Check the *Prop Get* radio button to indicate that this is a *Property Get* function and enter the name of the property (`Caption1`) to which it applies.

Note that it is not necessary for the property name referenced by the *Get* and *Put* functions to correspond to a variable name, although in this case it does.

The final *COM Properties* dialog box for `GetCaption1` should appear as follows. Click *OK* to save your changes.
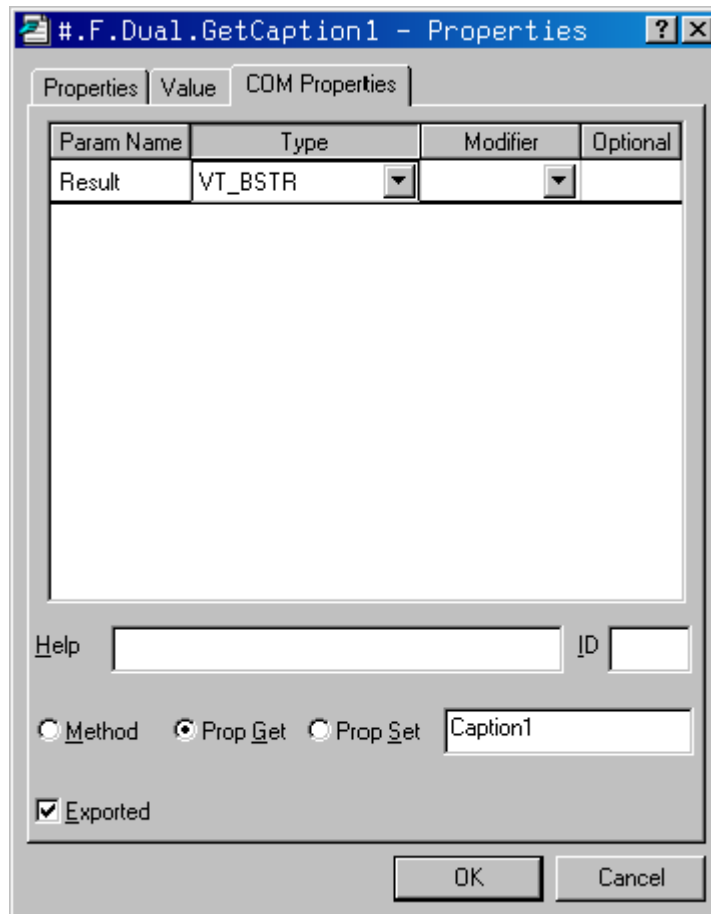
Now do the same for the `SetCaption1` function. This function takes an argument which it expects to be a character vector. It must therefore be defined as having a single parameter of data type *VT_BSTR*; the parameter name is unimportant. However, you must ensure that the *Optional* button is unchecked.

In APL terms, the function does not return a result. However, in OLE terms the result is defined to be of type *VT_VOID*. Alternatively, you may just leave this field empty.

The OLE properties for `SetCaption1` should appear as follows:
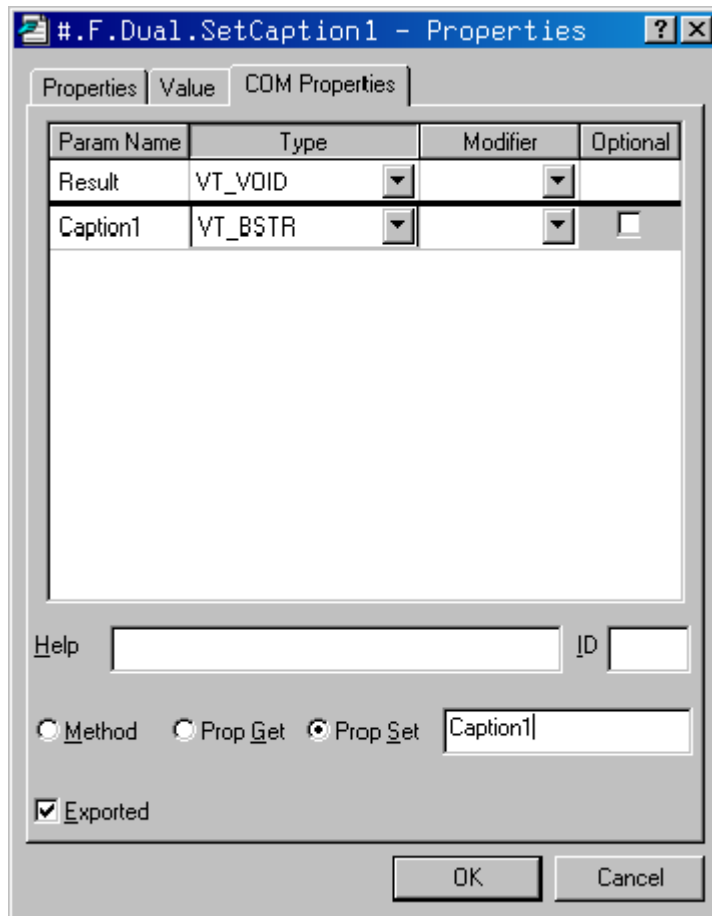
6. An alternative way to define the syntax for exported functions is to use the *COM Functions* tab in the Properties dialog box for the ActiveXControl object itself. (Hint: using the Workspace Explorer, open `F` so that its contents, `Dual`, are displayed in the right-hand list, select `Dual`, then click *Props*). The *COM Functions* tab should appear as follows:
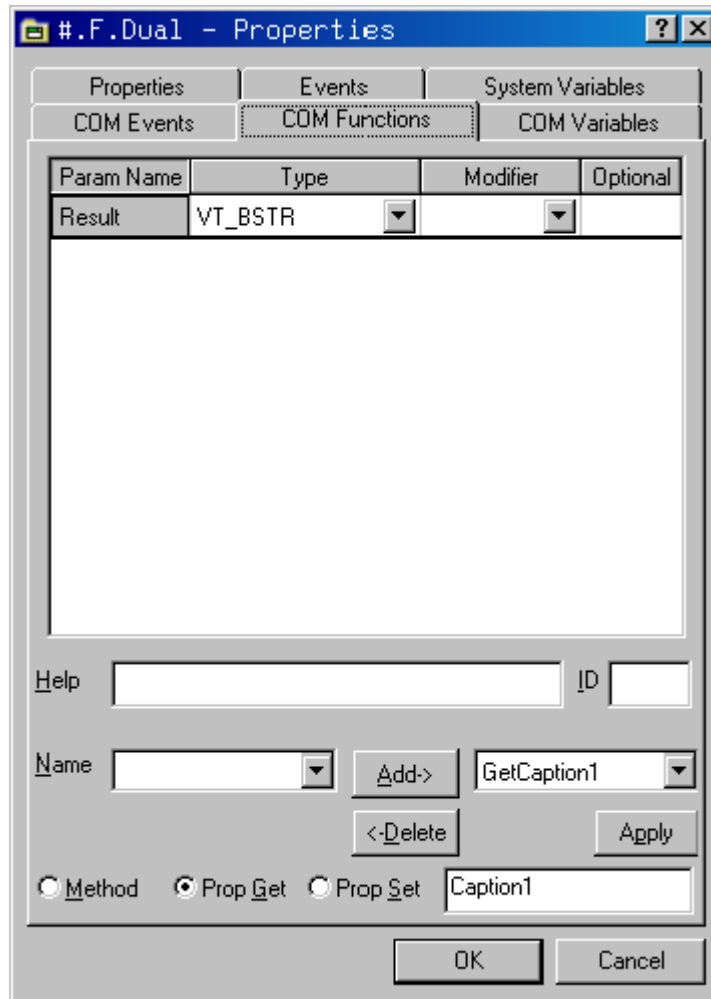
7. The right-hand Combo box allows you to view and edit their syntax for the exported functions you have already defined. The left-hand Combo box displays the list of other non-exported functions that are defined in the ActiveXControl.

8. Select `GetCaption2` from the left-hand Combo box, and then click *Add*. The dialog box will change to display the default syntax for `GetCaption2`. Alter the *Result* data type to VT_BSTR, select *Prop Get*, and enter the name of the property, `Caption2`, so that the dialog box appears as follows:
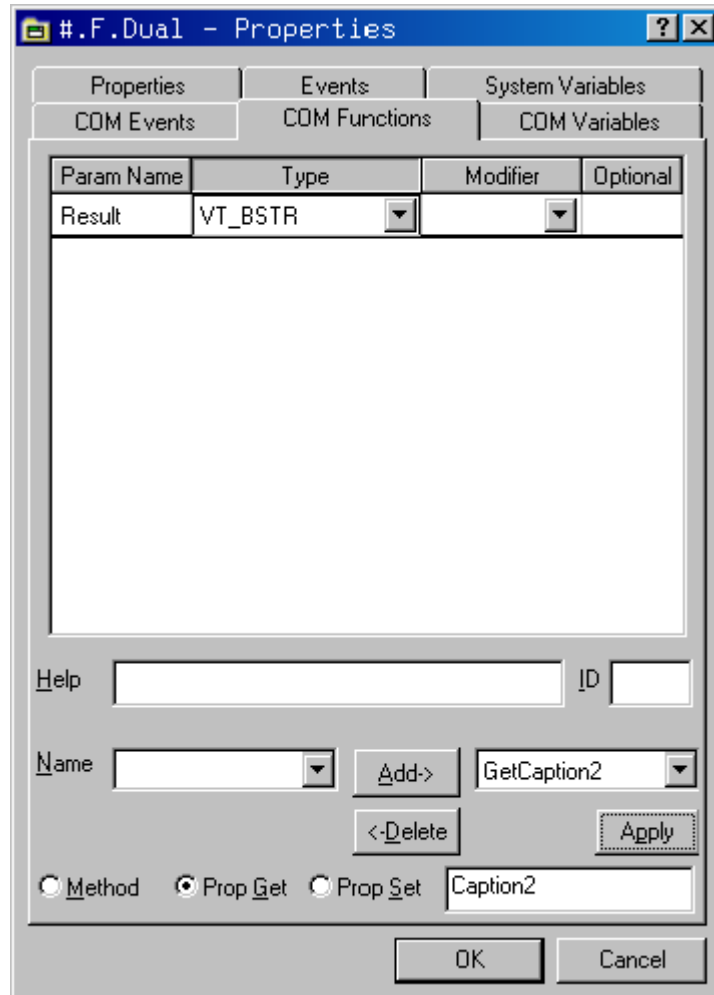
9. The third way to define the syntax for exported functions is to use the SetFnInfo method of the ActiveXControl object. This allows you to export functions using APL code, which in some circumstances may be more convenient than filling in dialog boxes.

The SetFnInfo method requires the name of the function, its syntax, a help id, a code which specifies its type (0 = method, 2 = property get, 4 = property put) and, if appropriate, the name of the property to which it applies, i.e.

```
SetFnInfo fn syntax helpid type property
```

The function *syntax* is a nested array whose first element defines the function's result and whose subsequent elements define each of its parameters. Each syntax specifier is a single character string that defines a data type, or a pair of character strings. If so, the first string for the result defines a help string, and the first string for each parameter defines its name.

The following table describes the information we must specify for each of the functions to be exported:

Exported Functions

Function	Result	Parameter		Get/ Put	Property
		Name	**Type**		
GetCaption1	VT_BSTR			Get (2)	Caption1
GetCaption2	VT_BSTR			Get(2)	Caption2
GetGradient	VT_R8			Get(2)	Gradient
GetIntercept	VT_R8			Get(2)	Intercept
GetValue1	VT_I4			Get(2)	Value1
SetCaption1	VT_VOID	Caption1	VT_BSTR	Put(4)	Caption1
SetCaption2	VT_VOID	Caption2	VT_BSTR	Put(4)	Caption2
SetGradient	VT_VOID	Gradient	VT_R8	Put(4)	Gradient
SetIntercept	VT_VOID	Intercept	VT_R8	Put(4)	Intercept
SetValue1	VT_VOID	Value1	VT_I4	Put(4)	Value1

From this table we can easily construct the corresponding SetFnInfo statements. For example, the statement for `GetCaption1` is:

```
SetFnInfo 'GetCaption1' 'VT_BSTR' ¯1 2 'Caption1'
```

Note that `¯1` in the 3rd element of the right argument specifies that there is no help id.

Open the function `F.Dual.EXPORT`; this contains statements to export all of the *Get* and *Put* functions we need.

Run the function and save the workspace.

```
      )CS
#
      F.Dual.EXPORT
      )SAVE
```

Then, re-export the workspace, updating your .OCX file with all the new information.

Setting Properties from Visual Basic

1. Close Dyalog APL

2. Start Visual Basic and click Standard EXE.

3. Click the right mouse button in the leftmost toolbox and select *Components* from the pop-up menu.

4. Locate the control named *Dyalog DUAL Objects*, set its check box on and click *OK*. This adds a tool for the Dual control to the Visual Basic designer toolbox.

5. Click on the new tool and drag out an outline on your Form. An instance of the Dual control (called Dual1) will appear.

6. In the Properties dialog box, set the Dual1 properties as follows:

Caption1	Centimetres
Caption2	Inches
Gradient	0.3937
Intercept	0

7. Repeat step 5 to position a second instance of the Dual control (Dual2) on your Visual Basic Form.

8. Double-click the left mouse button over your Form (*Form1*). This will bring up the code editor dialog box. Edit the Form_Load() subroutine, entering the following program statements. This code will be run when Visual Basic starts your application and loads the Form *Form1*. It illustrates how you can change the properties of your Dyalog APL ActiveX control dynamically.

```
Private Sub Form_Load()
Dual2.Caption1 = "Inches"
Dual2.Caption2 = "Centimetres"
Dual2.Intercept = 0
Dual2.Gradient = 2.54
End Sub
```

9. Now test your application by selecting Run/Start from the menubar. When you have finished, select Run/End.

10. Exit Visual Basic; do **not** save the Project or the Form.

Defining and Exporting Events

1. Start Dyalog APL and load the DUAL workspace (Hint: use the *File* menu; it will be the most recently saved file)

2. Using the Workspace Explorer, open the callback function `F.Dual ChangeValue` and alter `ChangeValue[2]` from:

    ```
     Value1←⊃¯1↑MSG
    ```
    to
    ```
     Value1←⊃4 ⎕NQ '' 'ChangeValue1' (⊃¯1↑MSG)
    ```

    Then close the function. Previously, the `ChangeValue` function simply accepted the new value (of the TrackBar thumb) that it received as the last element of the event message. Now it generates an external *ChangeValue1* event for the host application using `4 ⎕NQ`. The host may in turn modify the new value which is returned as the result of the expression. Thus not only can Dyalog APL generate events which are detectable by the host application, it can also accept modifications.

3. Again using the Workspace Explorer, open the Properties dialog box for the Dual object itself and select the *COM Events* tab.

    3.1. Enter the name of the event `ChangeValue1` into the edit box labelled *Name*.

    3.2. Click *Add*

    3.3. Click the right mouse button over the *Result* row and select *Insert*

    3.4. Change the name *Param1* to *Value1*, the *Type* to *VT_I4* and the *Modifier* to *VT_PTR*. This defines the event to supply a pointer to an integer. The fact that it is a *pointer* means that the (integer) parameter may be modified by the host application.

The final appearance of this dialog box should be as follows:
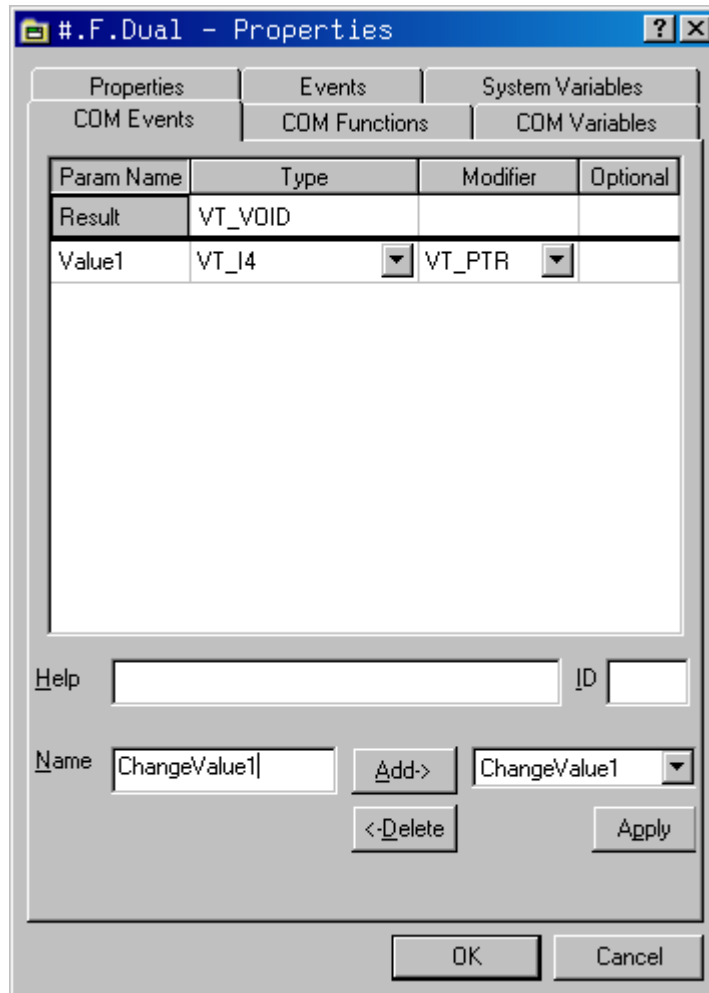
4. Click OK, change back to the root space, and save the workspace

    ```
    )CS #
    )SAVE
    ```

5. Select File/Export and rebuild your OCX file.

Using Events from Visual Basic

1. Close Dyalog APL

2. Start Visual Basic and click Standard EXE.

3. Click the right mouse button in the leftmost toolbox and select Components from the pop-up menu.

4. Locate the control named Dyalog DUAL Objects, set its check box on and click OK. This adds a tool for the Dual control to the Visual Basic designer toolbox.

5. Click on the new tool and drag out an outline on your Form. An instance of the Dual control (called Dual1) will appear.

6. Select the label tool and add a label object to the Form. Select its Font property and change it to 14-point bold.

7. Double-click over Dual1 to bring up the code window. Notice how Visual Basic presents you with a skeleton subroutine for the (only) event ChangeValue1 which we have just defined and exported. Notice too that Visual Basic knows that the single parameter is named Value1 and that its data type is Long (VT_I4).

8. Enter the following code, and then close the code window.

9. Private Sub Dual1_ChangeValue1(Value1 As Long)

10. Label1.Caption=Str(Value1)

11. End Sub

12. Start the application using Run/Start. Exercise the Dual control and observe that Visual Basic updates the Label1 control in response to the ChangeValue1 events. When you have finished, select Run/End.

13. Double-click over Dual1 to bring up the code window. Alter the Dual1_ChangeValue subroutine to the following, and then close the code window.

14. Private Sub Dual1_ChangeValue1(Value1 As Long)

15. Value1=2*(Value1\2)

16. Label1.Caption=Str(Value1)

17. End Sub

18. Start the application using Run/Start. Exercise the Dual control and observe that now the slider moves in increments of 2. When you have finished, select Run/End.

19. Close Visual Basic; do not save anything.

Using Dual in a Web Page

This part of the tutorial can be run using any Web Browser that supports ActiveX. These include Microsoft Internet Explorer and Netscape Navigator with the NCompass ScriptActive Plug-In.

1. Start Dyalog APL and load the DUAL workspace again.

   ```
   )LOAD DUAL
   )CS F.Dual
   ```

2. Look at the function `WRITE_HTML`. This function writes a very simple page of HTML that loads your Dyalog APL ActiveX control. The left argument to the function is a title for the page; the right argument is the full pathname for the file. The function references the control by embedding its ClassID in the HTML document.

3. Now run it:

   'Dyalog Dual Control' WRITE_HTML 'dual.htm'

4. Close Dyalog APL

5. Start your Web Browser and point it at the file you have just saved by typing the URL:

   file://c:\...dyalog...\dual.htm

6. Close your browser

Calling Dual from VBScript

In the last part of this tutorial, you will learn how you can manipulate the Dual control from VBScript in a web page.

For this example, we first need to export the Value2 property. This property is only required to be *read* (not set) by the VBScript program. Therefore there is no need for it to be accessed via Get and Put functions and it can be exported (more simply) as a variable.

1. Start Dyalog APL and load the DUAL workspace again.

   ```
   )LOAD DUAL
   ```

2. Using the Workspace Explorer, display the Object Properties dialog box for the variable Value2.

3. Select the COM Properties tab. As you have not yet defined any OLE attributes, the default display is as follows:
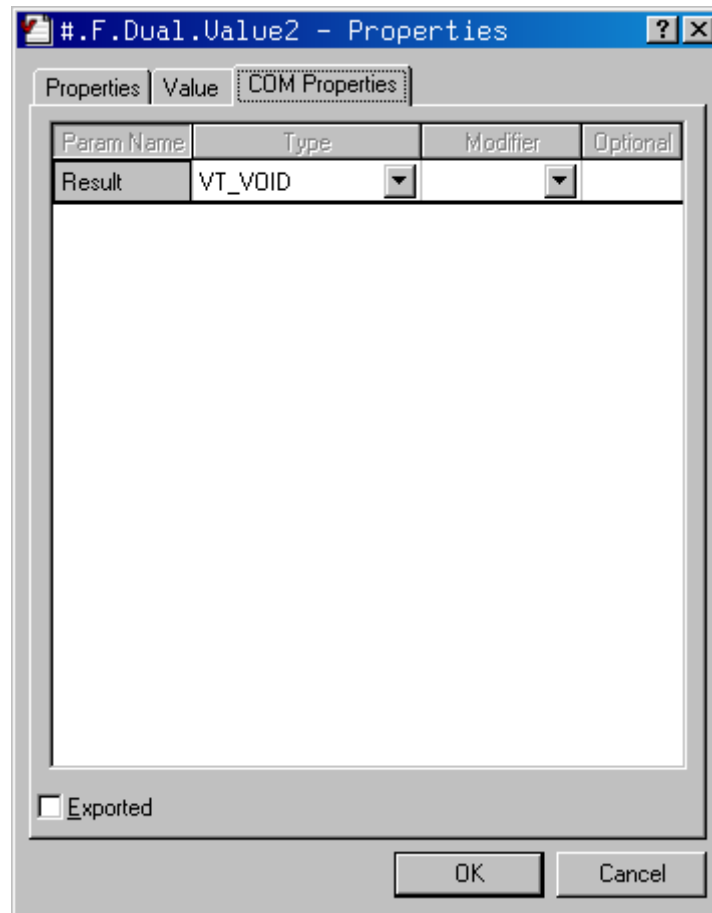
4. Check the *Exported* option button, and change the data type to VT_R8. This is important because unlike Value1, which is an integer, Value2 may be floating-point and it must be declared as such. The resulting dialog box should appear as below; click *OK* to save these settings.
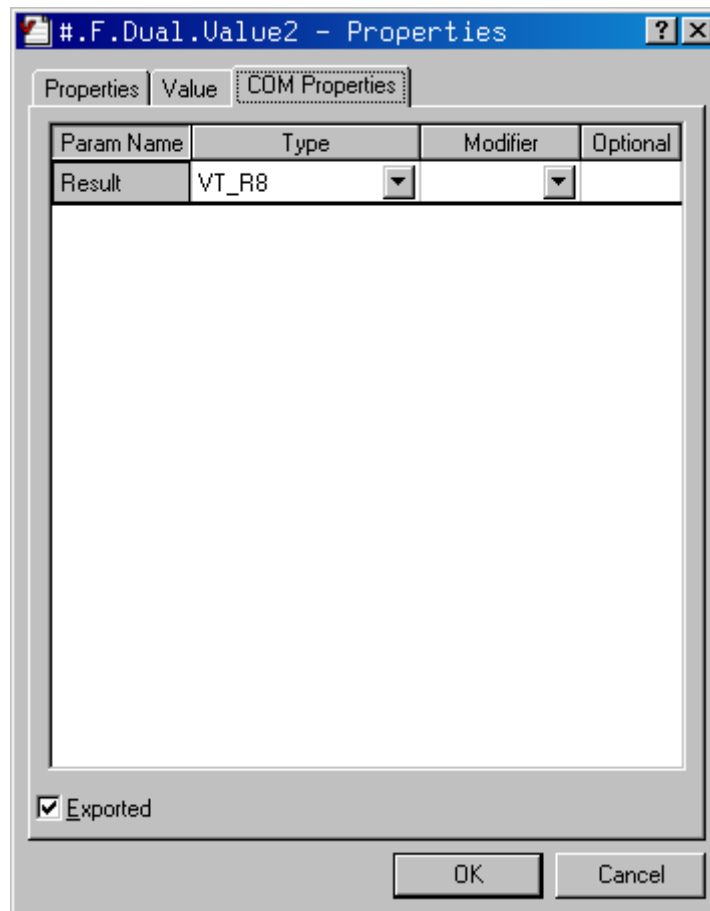
5. The VBScript program is going to need to know whenever the derived value, Value2, changes, Therefore, the next step is to define the code to generate a ChangeValue2 event and export it. ChangeValue2 is to be generated whenever Value2 has changed, so the place to put it is immediately after Value2 is recalculated in `CalcValue2`.

Edit `F.Dual.CalcValue2` so that it reads as follows:

```
     ∇ CalcValue2;SINK
[1]    Value2←Intercept+Gradient×Value1
[2]    :If ~(⊂'Create')∈⎕SI
[3]        SINK←4 ⎕NQ'' 'ChangeValue2'Value2
[4]    :EndIf
     ∇
```

Note that the function deliberately avoids generating the ChangeValue2 event when the instance is created. It can tell when this happens because during creation it will have been called by the Create function. (We could have instead have called CalcValue2 with an argument, but this will suffice.) This is necessary only because the simple VBScript example is unable to handle events during object creation

You can export the CalcValue2 event using the COM Events tab on the Properties dialog box for F.Dual. However, you can also export the event using the SetEventInfo method.

Type the following:

```
    INFO←'VT_VOID'('Value2' 'VT_R8')
    2 ⎕NQ'#.F.Dual' 'SetEventInfo' 'ChangeValue2' INFO
```

(You may wish to confirm that the event is registered correctly using the dialog box)

7. Save the workspace:

```
    )CS #
    )SAVE
```

8. Finally, you need to rebuild the OCX file to reflect these changes, so select File/Export and rebuild your OCX file.

9. The HTML page containing the example VBScript program is supplied in the file samples\activex\dualvb.htm. However, the Dual object to which the HTML refers (via its ClassID) is not the same object as *your* Dual object which has its own unique ClassID. We must update the file, changing the existing ClassID to the ClassID of your own Dual object. You can do this using the `UPDATE_CLASSID`.

10. Look at the function `F.Dual.UPDATE_CLASSID`. This function simply updates an HTML file with the ClassID of the current (ActiveXControl) namespace.

```
    ∇ {NEW}UPDATE_CLASSID FILE;NID;HTML;CLASSID;I
[1]    ⍝ Updates HTML file, replacing all object
[2]    ⍝ references with the ClassID of the current
[3]    ⍝ (ActiveXControl) namespace.
[4]    ⍝ Optional left argument is the name of the
[5]    ⍝ new HTML file. If omitted, it updates the
[6]    ⍝ file in-situ.
[7]
[8]    NID←FILE ⎕NTIE 0
[9]    HTML←⎕NREAD NID,82,(⎕NSIZE NID),0
[10]   :If 2=⎕NC'NEW'
[11]       ⎕NUNTIE NID
[12]       :Trap 22
[13]           NID←NEW ⎕NCREATE 0
[14]       :Else
[15]           NID←NEW ⎕NTIE 0
[16]       :EndTrap
[17]   :EndIf
[18]   I←'clsid:'∊{
[19]       ⎕AV[(⎕AV⍳⍵)-48×⍵∊⎕A]
[20]   }HTML
[21]   I←I/⍳⍴I
[22]   :If ×⍴I
[23]       I+←5
[24]       CLASSID←1↓¯1↓⎕WG'ClassID'
[25]       HTML[,I∘.+⍳⍴CLASSID]←((⍴I)×⍴CLASSID)⍴CLASSID
[26]       HTML ⎕NREPLACE NID 0
[27]   :EndIf
[28]   ⎕NUNTIE NID
    ∇
```

11. Now run the function, making a new dualvb.htm file in your current directory.

    ```
    )CS #.F.Dual
    'dualvb.htm' UPDATE_CLASSID 'samples\activex\dualvb.htm'
    ```

12. Close Dyalog APL.

13. Start your Web Browser and point it at the file you have just saved by typing the URL: `file://c:\...dyalog...\dualvb.htm`

14. The Web page displays two instances of your Dual control, one called *plank_length* labelled *Length* (Metres to Centimetres) and the other named *plank_width* and labelled *Width* (Inches to Centimetres). The initialisation of these controls is performed by the `window_onload()` which is run when the page is loaded into the Web Browser.

```
Sub window_onload()
plank_length.Caption1 = "Metres"
plank_length.Caption2 = "Centimetres"
plank_length.Gradient = 100
plank_length.Intercept = 0
plank_width.Caption1 = "Inches"
plank_width.Caption2 = "Centimetres"
plank_width.Gradient = 2.54
plank_width.Intercept = 0
end sub
```

15. Whenever you change one of these dimensions, the corresponding Dual control generates a ChangeValue2 event after the derived value (Value2) in centimetres is recalculated. Each of the Dual controls has a VBScript callback function attached which calculates the new area. These are as follows:

```
Sub plank_length_ChangeValue2(Value2)
Result.Plank_Area.value = Value2 *
plank_width.Value2
end sub

Sub plank_width_ChangeValue2(Value2)
Result.Plank_Area.value=Value2 * plank_length.Value2
end sub
```

16. When you have finished exercising the two Dual controls, close your Web Browser.

C H A P T E R 14

Shared Variables (DDE)

Introduction to DDE

Dynamic Data Exchange (DDE) is a protocol supported by Microsoft Windows that enables two applications to communicate with one another and to exchange data.

DDE has largely been superseded by COM, but continues to be supported by Dyalog APL for backwards compatibility. For new applications, use COM.

Two applications exchange information by having a **conversation**. In any conversation, there is a **client**, which is the application that initiates the conversation, and a **server**; the application that is responding to the client. An application may partake in several conversations at the same time, and may play the server role in some and the client role in others. Indeed, it is perfectly reasonable for two applications to have two conversations in which each acts as the server in one and the client in another.

Most conversations are effectively *one-way* in that data flows from the server to the client. However, conversations are potentially bi-directional and it is possible for the client to send data to the server. This is often described as *poking* data.

To initiate a DDE conversation, the client application must specify the name of the server and the subject of the conversation, called the **topic**. The combination of application and topic uniquely identifies the conversation. In most applications that support DDE, the topic is the "document name". For example, Microsoft Excel recognises the name of a spreadsheet file (.XLS or .XLC) as a topic.

During a conversation, the client and server exchange information concerning one or more **items**. An item identifies a particular piece of data. For example, Microsoft Excel recognises cell references (such as R1C1) as data **items** in a conversation. Throughout a conversation, the client may specify how it wishes to be updated when the data in the server changes. There are three alternatives. Firstly, the client can explicitly request the value of an item as and when it needs it. This is described as a **cold link**. Alternatively, a client may ask the server to supply it with the value of a particular item whenever its value changes. This is called a **hot link**.

Finally, it may ask the server to **notify** it whenever the value of an item changes, to which the client may respond by asking for the new value or not. This is termed a **warm link**.

In addition to providing a means for exchanging **data**, DDE provides a mechanism for one application to instruct another application to **execute** a command. This is implemented by sending a DDE_EXECUTE message. It is important to understand that the effect of the command is local to the application in which it is executed, and that the recipient of the message does **not** return a result to the originating application. It does **not** work like the APL execute function.

Shared Variable Principles

Shared Variables are part of the APL standard, although strictly speaking as an optional facility. They provide a comprehensive mechanism for communicating between two APL workspaces, or between APL and a co-operating non-APL application. Despite some conflicts between Shared Variable concepts and DDE, this standard APL mechanism has overriding advantages as the basis for a DDE interface. The main benefit is that Shared Variables provide a **general** basis for developing communications using a variety of protocols, of which DDE is but a single example. Dyalog APL communications are not therefore designed for and limited to DDE, but can be extended to other protocols which are appropriate in different environments.

Most mainframe APL users will already be familiar with Shared Variables and will need no introduction to their concepts. New APL users, or those whose experience has been only of PC-based interpreters, may find the following introduction helpful.

Introduction

It is easiest to consider Shared Variables between two APL workspaces. A Shared Variable is simply a variable that is common to and visible in two workspaces. Once a variable is shared, its value is the same in both workspaces. Communication is achieved by one workspace assigning a new value to the variable and then the other workspace referencing it. Although there is no explicit **send** or **receive**, it is perhaps easier to think of things in this way. When you assign a value to a shared variable, you are in effect transmitting it to your partner. When you reference a shared variable, you are in fact receiving it from your partner.

This discussion of shared variables will refer to the terms **set** and **use**. The term **set** means to assign a (new) value to a variable, i.e. its name appears to the left of an assignment arrow. The term **use** means to refer to the value of a variable, i.e. its name appears to the right of an assignment arrow.

Sharing a Variable

Variables are shared using the system function ⎕SVO. This is a dyadic function whose right argument specifies the name (or a matrix of names) of the variable, and whose left argument identifies the partner with whom the variable is to be shared. In mainframe APL, you identify the partner by its processor id. For example, the following statement means that you offer to share the variable X with processor 123.

        123 ⎕SVO 'X'

A single ⎕SVO by one workspace is not however sufficient to make a connection. It is necessary that **both** partners make an offer to share the variable. Thus if you are process 345, your partner must complete the coupling by making an equivalent shared variable offer, e.g.

        345 ⎕SVO 'X'

The coupling process is symmetrical and there is no specific order in which offers must be made. However, there is a concept known as the *degree of coupling* which is returned as the result of ⎕SVO. The degree of coupling is simply a count of the number of processes which currently have the variable "on offer". When the first process offers to share the variable, its ⎕SVO will return 1. When the second follows suit, its ⎕SVO returns 2. The first process can tell when coupling is complete by calling ⎕SVO monadically at a later point, as illustrated below.

Process 345	**Process 123**
`        123 ⎕SVO 'X'` `1`	
	`        345 ⎕SVO 'X'` `2`
`        ⎕SVO 'X'` `2`	

In this example, both partners specified exactly whom they wished to share with. These are termed **specific offers**. It is also possible to make a **general offer**, which means that you offer to share a particular variable with **anyone**. Coupling can be established by any other processor that offers to share the same variable with you, but notice that the other processor must make a **specific offer** to couple with your general one. The rule is in fact, that sharing may be established by matching a specific offer with another specific offer, or by matching a specific offer with a general offer. Two general offers cannot establish a connection.

The State Vector

One of the interesting things about Shared Variables, is that both APL workspaces are equal partners. Either of them is allowed to change the value of a shared variable, thus communication is two way. In any communication of this sort, it is essential to have a mechanism to keep things in step. If not, it is possible for one partner to miss something or to receive the same message twice. In some applications this doesn't matter. For example, if one APL workspace is simply monitoring the current value of a particular currency, it does not matter that a second workspace doesn't see all of the fluctuations as they occur. It is important only that the latest value can be referenced when it is needed. Contrast this with a trading application in which the trading workspace registers each transaction with a second workspace which monitors and stores the transactions on a database. Clearly in this case it is essential that each and every transaction is properly communicated and recorded.

Synchronisation is provided by two system functions, `⎕SVS` and `⎕SVC`. `⎕SVS` reports the current value of a shared variable's **State Vector**. This provides information concerning the state of the variable from each partner's point of view. The second function, `⎕SVC`, allows you and your partner to specify interlocking that enforces the level of synchronisation required by your application.

Each shared variable has a **state vector** which indicates which partner has set a value of which the other is still ignorant, and which partner is aware of the current value. The current state of a shared variable is reported by the monadic system function `⎕SVS`. Its argument is the name of the shared variable. Its result is a 4-element Boolean vector which specifies the current state vector, i.e.

```
state ← ⎕SVS name
```

The state vector will have one of the following values:

0 0 0 0	The variable is not shared
0 0 1 1	Both partners know the current value
1 0 1 0	You have set the value, but your partner has yet to use (read) it.
0 1 0 1	Your partner has set the variable but you have not yet used (read) it.

It may not be immediately apparent as to how the information provided by ⎕SVS can be used. The answer, as we will see later, is that communications generates **events**. That is to say, when your partner sets a shared variable to a new value or references a value that you have set, an event is generated telling you that something has happened. ⎕SVS is then used to determine what has happened (set or use) and, if you have several variables shared, which one of the variables has in some way changed state. A shared variable state change is thus the trigger that forces some kind of action out of the other process.

Access Control

⎕SVS is not sufficient on its own to synchronise data transfer. For example, what if the two partners both set the shared variable to a different value at **exactly** the same point in time ? This is the role of ⎕SVC which actually assures data integrity (if required) by imposing access controls. Its purpose is to synchronise the order in which two applications **set** and **use** the value of a shared variable.

In simple terms, ⎕SVC allows an application to inhibit its partner from setting a new value before it has read the current one, and/or to inhibit its partner from using a variable again before it has been reset.

⎕SVC is a dyadic system function. Its right argument specifies the name of the shared variable; its left argument the access control vector, i.e.

```
access ⎕SVC name
```

The access control vector is a 4-element Boolean vector whose elements specify access control as follows:

[1]	1 means that you cannot set the variable until your partner has used it.
[2]	1 means that your partner cannot set the variable until you have used it.
[3]	1 means that you cannot use the variable until your partner has set it.
[4]	1 means that your partner cannot use the variable until you have set it.

In principle, each of the two partners maintains its own copy of the access control vector using ⎕SVC. Control is actually imposed by the **effective access control vector** which is the result of "ORing" the two individual ones. From your own point of view, the effective access control vector is:

```
(your ⎕SVC)  ∨  (your partner's ⎕SVC)[3 4 1 2]
```

Whenever either of the partners attempts an operation (set or use) on a shared variable, the system consults its effective access control vector. If the vector indicates that the operation is currently permitted, it goes ahead. If however the vector indicates that the operation is currently inhibited, the operation is delayed until the situation changes.

For example, suppose that the effective access control vector is (1 0 0 1). This prevents either partner from setting the shared variable twice in a row, without an intervening use by the other. The purpose of this is to prevent loss of data. Suppose now that one workspace assigns the value 10 to the shared variable (which is called `DATA`), i.e.

```
      DATA ← 10
```

Then, before the partner has referenced the new value it attempts to execute the statement:

```
      DATA ← 20
```

APL will **not** execute the statement. Instead it will wait (indefinitely if required) until the partner has received the first value (10). Only then will the second assignment be executed and processing continued. Effectively one workspace stops and waits for the other to catch up.

Similarly, suppose that the effective access control vector is (0 0 1 1). This means that neither partner can **use** the variable twice in succession without an intervening **set** by the other. This type of control is appropriate where each **set** corresponds to an individual transaction, and you want to prevent transactions from inadvertently being duplicated.

Suppose now that one workspace references the shared variable (which is called `DATA`), i.e.

```
      TRANSACTION ← DATA
```

Then, soon after, it executes the statement again, but without an intervening **set** by its partner, i.e.

```
      TRANSACTION ← DATA
```

This time, the reference to `DATA` is inhibited, and the workspace waits (indefinitely if necessary) until the partner has assigned a new value. Only then will the second reference be executed and processing continued. Again, one workspace stops and waits for the other.

The purpose of ⎕SVC is to synchronise data transfer. It is particularly useful where timing considerations would otherwise cause data loss. However, an incorrect application which makes inappropriate use of ⎕SVC may hang.

A second type of problem can occur during the development of an application that uses shared variables. If the program is interrupted by an error, an attempt to display the value of a shared variable counts as a "use" and, if inhibited, will hang. In applications that use interlocking, it is recommended that a shared variable is explicitly "used" by making an assignment to a temporary variable which can then be referenced freely.

This is the theory; we will now see how DDE, by its very nature, imposes certain limitations in practice.

APL and DDE in Practice

The interface between Dyalog APL/W and DDE is provided by Shared Variables which are implemented as closely as possible in accordance with the APL Standard. There are however some conflicts between Shared Variables and the way in which DDE works. These impose certain restrictions.

The APL Shared Variable concept is based upon the *peer-to-peer* communications model where each partner has equal rights and equal control. DDE however is based upon the *client-server* model whereby data (normally) flows from server to the client at the client's request. This in turn has two major implications. Firstly, a client must **initiate** a DDE conversation. A server may only respond to a request from a client for a connection; it may not itself start a conversation. Secondly a server cannot specify to which client it wishes to communicate. In terms of the APL standard, this means that if a shared variable is to act as a server it must be made the subject of a *general offer*. A shared variable that is to act as a client must be the subject of a *specific offer* Furthermore, as in any DDE conversation there must be one server and one client, it means that two APL workspaces can share variables only if one makes a general offer and one makes a specific offer.

An APL application registers itself as a potential server, or initiates a DDE conversation as a client, by making a Shared Variable offer using ⎕SVO. The offer is either a general offer, which corresponds to a DDE server, or a specific offer which is a client.

Note that, as mentioned in the introduction, DDE does not preclude two-way data transfer, despite its insistence on a client-server relationship. Thus the establishment of a shared variable as a server or as a client does not force the data transfer to be one-way. The choice of whether APL is to act as a server or as client may in practice be determined by convenience.

APL as the Client

To initiate a DDE conversation with a server, you use `⎕SVO` as follows:

```
'DDE:appln|topic' ⎕SVO 'var item'
```

where:

`appln`	is the name of the server application.
`topic`	is the server topic (usually the name of a document).
`var`	is the name of the APL variable.
`item`	is the name of the item with which the variable is to be associated (shared).

For example, the following statement would associate the variable `SALES` with the block of cells R1C1 to R10C10 in an Excel spreadsheet called "Budget".

```
'DDE:EXCEL|BUDGET' ⎕SVO 'SALES R1C1:R10C10'
2
```

Note that the result of `⎕SVO` is the *degree of coupling*. This has the value 2 if the connection is complete (the server has responded) and 1 if it has not. In practice it is a little more complicated than this, because the result actually depends upon the type of DDE link that has been established.

In principle, the type of link is determined by the client. However, because the server may refuse to accept a particular type of link, it can actually be a result of *negotiation* between the two applications.

When the shared variable is offered as a client, APL **always** requests a warm link from the server. If the server refuses a warm link, APL instead requests the current value of the data item (a cold link), and, if the server responds, APL stores the value in the variable. In either case, the degree of coupling is set to 2 if the connection was successful.

Executing Commands in the Server

As mentioned in the Introduction, it is possible for a client to instruct a server to execute a command by sending it a DDE_EXECUTE message. This is intended to allow the client to condition the environment in which the server is operating and not (as one might first expect) to execute a command which directly returns a result. In fact the only response from a server to a DDE_EXECUTE message is a positive or negative acknowledgement, the meaning of which is application dependent.

You can establish a shared variable as a channel for sending DDE_EXECUTE messages by assigning it a surrogate name of '⍎', the APL execute symbol. After sharing, you send commands to the server as DDE_EXECUTE messages by assigning them, as character vectors, to the shared variable. Following each such assignment, the value of the shared variable is reset to 1 if the server responded with a positive acknowledgement, or 0 if it responded with a negative acknowledgement. This should be interpreted with reference to the server application documentation. Note that most applications require that commands are surrounded by square brackets but several commands may be sent at a time. The following examples use Microsoft Excel Version 2.0 as the server :

Establish a link to Excel's SYSTEM topic :

```
      'DDE:EXCEL|SYSTEM' ⎕SVO 'X ⍎'
2
```

Instruct EXCEL to open a spreadsheet file :

```
      X←'[OPEN(c:\mydir\mysheet.xls)]'
      X
1
```

Instruct EXCEL to select a range of cells :

```
      X←'[SELECT("R1C1:R5C10")]'
      X
1
```

Carry out two commands in one call :

```
      CMD1←'[OPEN(c:\mydir\mysheet.xls)]'
      CMD2←'[SELECT("R1C1:R5C10")]'
      X←CMD1,CMD2
      X
1
```

APL as the Server

A DDE conversation is initiated by a client, and not by a server. If you wish to act as a server, it is therefore necessary to register this fact with the APL interpreter so that it will subsequently respond to a client on your behalf. This is done by making a *general offer* using ⎕SVO as follows:

```
'DDE:' ⎕SVO 'var item'
```

where:

`var`	is the name of the APL variable.
`item`	is the name of the item with which the variable is to be associated (shared).

Notice that in this case, the left argument to ⎕SVO specifies only the protocol, 'DDE'. APL automatically defines the application name and topic to be 'DYALOG' and ⎕WSID respectively. The DDE *item* is specified in the right argument as either the name of the variable, or, optionally, as its external name or surrogate.

To allow another application to act as a client, you must have previously published the name(s) of the items which are supported. For example, if your APL application provides SALES information, the following statement could be used to establish it as a server for this item:

```
'DDE:' ⎕SVO 'X1 SALES'
1
```

In the case of a single general offer, the result of ⎕SVO will always be 1. When subsequently a client application attempts to initiate a conversation with a server with the application name 'DYALOG' and topic ⎕WSID, the APL interpreter will respond and complete the connection.

At this point, if and when the client has requested a hot or warm link to the item *SALES*, the degree of coupling (which is reported by using ⎕SVO monadically) becomes 2, i.e.

```
⎕SVO 'X1'
2
```

State and Access Control

Earlier, we have seen how shared variable state and access controls are used to ensure effective communication between two APL tasks. How do these concepts apply in the DDE environment when APL is using shared variables to communicate via DDE with both other APL workspaces, and with non-APL applications?

The initial state of a shared variable on the completion of sharing depends upon whether your variable is a server or a client. If it is a server, the initial state vector is (1 0 1 0) which means that you have set (and know) the value, but your partner has yet to use it. If the variable is acting as a client, the initial state vector is (0 1 0 1). This implies that your partner has set the value but you have yet to use it.

As your partner can be a non-APL application which does not share the concepts of **set** and **use**, it is necessary to define a rule or set of rules from which APL can reasonably infer such actions.

During a DDE conversation, the physical transfer of data from one application to another is achieved using DDE DATA messages. When a DATA message is sent, the receiving task normally returns an ACK (acknowledgement) message. APL uses the DATA and ACK messages to control Shared Variable access.

When an assignment is made to a shared variable, APL sends a DATA message to the second process. When it receives back an ACK message, APL infers that this means that the partner has **used** the variable. When APL receives a DATA message from the other process it infers that the partner has **set** the variable. However, it only responds with an ACK message when the new value of the variable is referenced by the workspace.

Let's see what this means if two APL workspaces are involved.

Server Workspace	Client Workspace

Make general offer
 	
`      X←42`
 	
`      'DDE:' ⎕SVO 'X'`
`1`
 	
`      ⎕SVS 'X'`
`0 0 0 0 ⍝ No partner`
 	
`      ⎕SVC 'X'`
`0 0 0 0 ⍝ No access control`	

`ack --->`
 	
`ack --->`
 	
`      ⎕SVS 'X'`
`1 0 1 0⍝ I know, he doesn't`
	Make specific offer
	`'DDE:DYALOG|SERVER' ⎕SVO 'X'`
	`<--- initiate ---`
	
	`<--- please advise on change`
	
	`2  ⍝ Offer accepted`
	`      ⎕SVS 'X'`
	`0 1 0 1⍝ He knows, I don't`

`--- data (42) --->`
 	
`      ⎕SVS 'X'`
`0 0 1 1⍝ We both know`
	Client requests data
	`        Y ← X`
	`<--- req ---`
	
	`<--- ack ---`
	`      ⎕SVS 'X'`
	`0 0 1 1⍝ We both know`

Server changes data
`      X ← 20`
`--- data has changed --->`
 	
`      ⎕SVS 'X'`
`1 0 1 0⍝ I know, he doesn't`
	
	`<--- ack ---`
	`      ⎕SVS 'X'`
	`0 1 0 1⍝ He knows, I don't`

`--- data (20) --->`
 	
`      ⎕SVS 'X'`
`0 0 1 1⍝ We both know`
	Client requests data
	`        Y ← X`
	`<--- req ---`
	
	`<--- ack ---`
	`      ⎕SVS 'X'`
	`0 0 1 1⍝ We both know`

As you can see, this has the desired effect, namely that an APL workspace sets the value of a shared variable by assignment to it and **uses** it by reference to it. The mechanism of using the DATA and ACK messages to imply **set** and **use** also works with non-APL applications which do not (in general) support these concepts.

Access control between two APL workspaces is imposed by each workspace acting independently. Whenever either workspace changes its ⎕SVC, the information is transmitted to the other. Thus both workspaces maintain their own copy of the **effective** access control vector upons which to base decisions.

Server Workspace	Client Workspace
No access control ⎕SVC 'X' 0 0 0 0 ⍝ No access control	**No access control** ⎕SVC 'X' 0 0 0 0 ⍝ No access control
	Client makes multiple requests for data Y←X Y←X
Server can set several times X←30 X←40	
Set access control 1 0 0 1 ⎕SVC 'X' --- change in ⎕SVC --> ⎕SVC 'X' 1 0 0 1⍝ I cannot set until he has used; he cannot use until I have set	⎕SVC 'X' 0 1 1 0⍝ He cannot set until I have used; I cannot use until he has set
	Client requests data Y ← X <--- req --- (hangs waiting for data)
Server changes data X ← 30 --- data (30) --->	
	<--- ack --- Y⍝ data received 30

Server Workspace	Client Workspace

Server changes data
 	
     `X ← 40`
      `--- data has changed --->`
 	
Server tries to change data again
 	
     `X ← 50`
      `--- data has changed --->`
(assignment hangs waiting for ack)
 	
          `--- data (40) --->`
 	
     `X⍝ assignment done`
50
	
	`<--- ack ---`
	
	          `Y ← X⍝ use data`
	`<--- req ---`
	
	`<--- ack ---`
	     `Y⍝ data received`
	40

Where the second process is a non-APL application, the effective access control vector is maintained only by the APL task and access control can only be imposed by APL. At first sight, it may seem impossible for APL to affect another application in this way, and indeed there are severe limitations in what APL can achieve. Nevertheless, effective access control is possible in the case when it is desirable to inhibit the partner from **setting** the value twice without an intervening **use** by the APL task.

This is simply achieved by withholding the ACK message. Thus if APL receives a DATA message from its partner at a time when a **set** by the partner is inhibited, APL registers the new value but withholds the acknowledgement. Only when the inhibitor is removed will APL respond with an ACK. (Users with DDESPY will observe that this is actually implemented by APL re-transmitting the DATA message to itself when the inhibitor is removed).

Assuming that the second application waits for the acknowledgement before proceeding, this will cause the desired synchronisation. Naturally, this cannot be entirely guaranteed because APL has no **direct** control over a non-APL program. Indeed, when an application transmits a DATA message, it can include a flag to indicate that an acknowledgement is neither expected nor required. In these circumstances, APL is powerless to impose any access control.

Note that APL does not (and cannot) have any control over successive internal references to the data by a non-APL application.

The rule for establishing your partner's initial ⎕SVC is as follows :

- If the DDE link is a **warm** link, your partner's ⎕SVC is initially (0 0 0 0).

- If the DDE link is instead a **hot** link, your partner's ⎕SVC is initially (1 0 0 1).

This works in practice as follows :

Server = APL, Client = APL

You made a general offer which has been accepted by another APL workspace, e.g.

```
    'DDE:' ⎕SVO 'DATA'
```

Two APL tasks always use a warm DDE link. Therefore, initially, both ⎕SVCs are (0 0 0 0). Control is (optionally) imposed by both partners subsequently setting ⎕SVC.

Server = APL, Client = another application

You made a general offer which has been accepted by another application, e.g.

```
    'DDE:' ⎕SVO 'DATA'
```

The client application establishes the strength of the link (warm or hot). If it is a warm link, the initial value of the client's ⎕SVC is (0 0 0 0) and, as the client has no means to change it itself, control may only be imposed by the server APL task. If the client establishes a hot link, its initial ⎕SVC is (1 0 0 1). As it has no means to change it, and as the APL server task cannot (by definition) change it, the client's ⎕SVC retains this setting for the duration of the conversation. (1 0 0 1) means that both partners are inhibited from setting the value of the shared variable twice in a row without an intervening use (or set) by the other. Given that the other application has requested a hot link (give me the value every time it changes) it is reasonable to assume that the application does not want to miss any values and will happily accept new data every time it is changed.

Server = another application, Client = APL

You made a **specific offer** to another application, e.g.

```
    'DDE:EXCEL|SHEET1' ⎕SVO 'DATA R1C1:R3C4'
```

In this case, APL as the client will request a warm DDE link. If the server fails to agree to this request, APL will ask for the current data value and, whether or not the server responds, will not establish a permanent link. Thus the only possibility for a permanent connection is a warm link. This in turn means that the server's ⎕SVC will be (0 0 0 0). Furthermore, as the server has no means to change it, it's ⎕SVC will remain (0 0 0 0) for the duration of the conversation. Control is therefore imposed solely by APL.

Terminating a Conversation

A DDE conversation is terminated by "un-sharing" the variable. This can be done explicitly using ⎕EX or ⎕SVR. It is also done automatically when you exit a function in which a shared variable is localised.

Example: Communication Between APLs

The following instructions will allow you to explore how the DDE interface can be used to communicate between two Dyalog APL/W workspaces.

1.   Start two separate APL sessions and arrange their windows one above the other so that they do not overlap.

2.   Select the top window and type :

```
      )WSID SERVER
      A←?5 5⍴100 ◇ A
      'DDE:' ⎕SVO 'A EXTNAME'
1
```

The result of ⎕SVO is 1, indicating that no client has yet joined in the conversation.

3.   Select the lower window and type :

```
      )WSID CLIENT
      'DDE:DYALOG|SERVER' ⎕SVO 'B EXTNAME'
B
```

The result of ⎕SVO is 2 indicating that the connection with the SERVER workspace has been successfully made. Now type B. It will have the same value as A in the upper window.

4.   Select the top window (SERVER) again and type :

```
      A←⌽A
      ⎕SVS 'A'
1 0 1 0
```

Note that the result of ⎕SVS indicates that the SERVER has set A, but the CLIENT has not yet referenced the value.

5.   Select the lower window (CLIENT) and type :

```
      ⎕SVS 'B'
0 1 0 1
      B
...
      ⎕SVS 'B'
0 0 1 1
```

Note how, after referencing the shared variable, its state has changed.

6.   Still in the CLIENT workspace, write the following function called `FOO`:

```
      ∇ FOO
[1]   ⍝ This function gets called on event 50 (DDE)
[2]    →(0 0 1 1≡⎕SVS'B')/0 ⍝ Exit if no change
[3]    B
      ∇
```

Then, to attach `FOO` as a callback and to "wait"...

```
      '.' ⎕WS 'Event' 50 'FOO'
⎕DQ '.'
```

7.   Now switch to the upper window (SERVER) and type :

```
      A←⌷A
```

       Type this expression repeatedly, or experiment with others. Note how changing `A` generates a DDE event (event number 50) on the system object `'.'` in CLIENT, which in turn fires the callback.

10.   To interrupt `⎕DQ` in the CLIENT, type Ctrl+Break or select "Interrupt" from the *Action* menu in the Session Window.

Example : Excel as the Server

The following instructions will allow you to explore the DDE interface with another application (in this case Microsoft Excel) acting as the server.

1.   Start Excel and enter some data into (say) the cells R1C1 to R4C3 of the spreadsheet "SHEET1". The data can be character strings and/or numbers. Note that if the spreadsheet is NOT called "SHEET1", the function RUN   below should be changed accordingly.

2.   Start Dyalog APL/W (clear ws).

3.   Size your windows so that both the Excel window and the APL Session window can be viewed comfortably at the same time.

4.   Type the following statement in the APL Session :

```
        'DDE:EXCEL|SHEET1' ⎕SVO 'X R1C1:R4C3'
```

    2

        The result should be 2. If not, please check that you have typed the expression correctly, and that the name of the topic (SHEET1) corresponds to the spreadsheet name displayed by Excel.

        Note that the character between "EXCEL" and "SHEET1" may be the ASCII *pipe* symbol or the APL stile. Also note that in some countries, you use Lnn instead of Rnn to refer to rows in Excel. You may therefore need to use the following expression instead:

```
        'DDE:EXCEL|SHEET1' ⎕SVO 'X L1C1:L4C3'
```

    2

5.   Remaining in the APL Session, type X. It is a matrix containing as many cells as you have requested in the ⎕SVO statement. If you entered any character strings, X will be nested.

6.   Switch to your Excel window and change the data in one or more of the cells.

7.   Switch back to the APL Session and look at X again. It will contain the new data.

8.   Look at the state of the shared variable X using ⎕SVS. It indicates that both partners are aware of the current value of X.

```
        ⎕SVS 'X'
0 0 1 1
```

9.   Now switch to Excel and change the data again. Repeat step 8. Note the result indicates that Excel has changed X, but you have not yet referenced it.

```
        ⎕SVS 'X'
0 1 0 1
```

10.   Type the expressions :

```
        '.' ⎕WS 'EVENT' 50 1
        ⎕DQ'.'
```

11.   Now switch to Excel and change the data again. Note that the ⎕DQ terminates and returns a result.

```
      .  50
```

12.   Switch back to APL and create the following function :

```
      ∇ FOO MSG
[1] 'MSG IS ' MSG
[2] 'X IS' X
      ∇
```

Then type :

```
        '.' ⎕WS 'EVENT' 50 'FOO'
        ⎕DQ'.'
```

13.   Now switch back to Excel and change the data. Note that every time you change a cell, the DDE event fires your callback function FOO. In fact the function is fired twice because it itself alters the STATE of X by *referencing* it. This causes a second DDE event.

14.   Switch back to APL, and type Ctrl+Break or select "Interrupt" from the *Action* menu to interrupt ⎕DQ.

Example : Excel as the Client

The following instructions will allow you to explore the DDE interface with APL acting as the server to another application; in this case Microsoft Excel.

1.   Start APL (clear ws) and type the expressions :

```
)WSID MYWS
X←12
'DDE:' ⎕SVO 'X SALES'
```

The workspace MUST have a name as this is broadcast as the DDE **topic**. Note that it is currently essential that X contains a value before you make the offer. The result of ⎕SVO is 1, indicating that no client has yet joined in the conversation.

2.   Start Excel (empty spreadsheet).

3   Size your windows so that both the Excel window and the APL Session window can be viewed comfortably at the same time. Do NOT iconify either one.

4.   Select the Excel window and type the following formula into the first cell :

=dyalog|myws!sales

the value of X (12) will now appear in the cell.

5.   Switch to the APL Session and type :

```
⎕SVO'X'
2
```

Notice that now that Excel has made the connection, the degree of coupling is 2.

6.   Now type :

```
X←34
```

You will immediately see the new value appear in your spreadsheet.

7.   Create the following function in your workspace :

```
      ∇FOO MSG
[1]   MSG
[2]   X←⎕AI[2]
      ∇
```

Then type the expressions :

```
'.' ⎕WS 'EVENT' 50 'FOO'
⎕DQ '.'
```

8.   The link between Excel and APL is a *warm* link (the type of link is determined by the client, so other applications may behave differently). This means that APL will send the new value of X (SALES) to Excel every time it changes. If you have DDESPY.EXE, you can verify what is happening.

9.   To interrupt ⎕DQ, type Ctrl+Break or select "Interrupt" from the *Action* menu in the Session Window.

Example : APL as Compute Server for Excel

The following instructions illustrate how APL can act as a "compute server" for Microsoft Excel, using two shared variables. One variable is used to read the data from Excel; the other is used to pass back the result.

1.   Start Excel and enter some NUMBERS into the cells R1C1 to R3C3 of the spreadsheet "SHEET1".

2.   Start Dyalog APL/W and size your windows so that both the Excel window and the APL Session window can be viewed comfortably at the same time.

3.   )LOAD the EXCEL workspace. This contains the following functions :

```
      ∇ RUN;Z;⎕WSID
[1]     Z←'DDE:EXCEL|SHEET1'⎕SVO 'DATA R1C1:R3C3'
[2]     →(2=Z)/L1
[3]     'No Excel out there ?' ◊ →0
[4]    L1:
[5]     CALC
[6]     ⎕WSID←'EXCEL'
[7]     Z←'DDE:'⎕SVO 'RESULT ANSWER'
[8]     'Now type "=dyalog|excel!answer" into'
[9]     'cell A4 in your spreadsheet'
[10]   L2:⎕DL 1
[11]    →(2≠⎕SVO 'RESULT')/L2 ⍝Wait for Excel to connect
[12]    'Connected ...'
[13]    '.'⎕WS 'EVENT' 50 'CALLB'
[14]   ⎕DQ '.'
      ∇


      ∇ CALLB MSG
[1]    ⍝ Callback to recalculate when Excel changes DATA
[2]    →(0 0 1 1≡⎕SVS 'DATA')/0
[3]     CALC
      ∇


      ∇ CALC;⎕TRAP
[1]     ⎕TRAP←0 'C' '→ERR'
[2]     RESULT←+/,DATA
[3]     →0
[4]    ERR:RESULT←⊂⎕EM ⎕EN
      ∇
```

4.   Type the following statement in the APL Session :

                    RUN

           Now type "=dyalog|excel!answer" into cell A4 in your spreadsheet

5.   Follow the above instructions to establish a link from APL to cell A4 in your
           Excel spreadsheet. The result of the computation will be displayed.

6.   Try changing some of the numbers in the spreadsheet and watch as APL re-
           calculates the sum.

7.   Try entering a character string in cell A1. Note that APL sends back a
           character string containing DOMAIN ERROR.

8.   Use Ctrl+Break or select "Interrupt" from the *Action* menu in the Session
           window to stop ⎕DQ.

Example : Using an Excel Command Macro

1. Start Excel, and open the sample worksheet called "EXCEL\APLCOMP.XLS" :

Quarter 1	Quarter 2	Quarter 3	Quarter 4
123	345	234	245
213	324	345	345
123	453	678	567

and the sample macro "WDYALOG\EXCEL\APLCOMP.XLM":

	A	B
1		APL_Total
2	total_channel	=INITIATE("dyalog", "excel")
3	total_type	=1
4		=POKE(total_channel,'CTL',total_type)
5		=TERMINATE(total_channel)
6		=RETURN()
7		
8		APL_Mean
9	mean_channel	=INITIATE("dyalog","excel")
10	mean_type	=2
11		=POKE(mean_channel,'CTL',mean_type)
12		=TERMINATE(mean_channel)
13		=RETURN()
14		
15		APL_Quit
16	quit_channel	=INITIATE("dyalog","excel")
17		=3
18		=POKE(total_channel,'CTL',total_type)
19		=TERMINATE(total_channel)
20		=RETURN()
21		

These macros may be run from the main Macro Menu, or by using these short-cut keys from within the spreadsheet:

Ctrl+a   Total

Ctrl+b   Mean

Ctrl+c   Quit

2.   Invoke APL and load the sample workspace EXCEL. This contains the
          following functions:

```
     ∇ LX;CTL;SVO
[1]    ⍝ Compute server for Excel
[2]    ⍝ Ensure workspace name is EXACTLY correct
[3]     ⎕WSID←'EXCEL'
[4]    ⍝ Set control variable
[5]     CTL←0
[6]    ⍝ Make a general offer to anyone
[7]      SVO←'DDE:'⎕SVO'CTL'
[8]    ⍝ On a DDE event, call function COMPUTE
[9]     '.'⎕WS'EVENT' 50 'COMPUTE'
[10]   ⍝ Enter the DQ
[11]    ⎕DQ'.'
     ∇
```

```
     ∇ COMPUTE;SVO;TYPE;DATA;RESULT
[1]    ⍝ Perform required calculation when user
[2]    ⍝ POKES a new value for CTL from Excel
[3]    ⍝ CTL is a matrix; ravel and assign to TYPE, so
          that a DDE message isn't generated
[4]     →0×⍳0=TYPE←,CTL ◇ →L3×⍳3=TYPE
[5]    ⍝ Setup shared variables
[6]     SVO←'DDE:EXCEL|APLCOMP.XLS'⎕SVO'DATA R2C1:R4C4'
[7]     SVO←'DDE:EXCEL|APLCOMP.XLS'⎕SVO'RESULT R1C5:R4C5'
[8]    ⍝ Which calculation?
[9]     →(L1,L2)[TYPE]
[10]   ⍝ Total
[11]  L1:RESULT←(4 1)⍴(⊂'Total'),+/DATA ◇ →L4
[12]   ⍝ Mean
[13]  L2:RESULT←(4 1)⍴(⊂'Mean'),(+/DATA)÷⊃⍴DATA ◇ →L4
[14]   ⍝ Stop the engine
[15]  L3:⎕OFF
[16]   ⍝ Reset control variable
[17]  L4:CTL←0
     ∇
```

3. In the APL window, trace `LX` until you are in the `⎕DQ`. Switch to the spreadsheet window, and from the main menu, select Macro, Run, APL_Total, Step, then Evaluate each step.

    Check that the channel is open, and that the POKE is successful. If the POKE is successful, you will see the trace window for COMPUTE pop-up. Complete the macro steps, then switch to the COMPUTE trace window. Trace through each line, and you should see the calculated totals appear in Excel.

    Continue tracing (COMPUTE will be called again in response to the change in the control variable CTL) until you are again in the `⎕DQ`.

4. Try invoking the APL_Mean macro now by typing the short-cut key **Ctrl+b** in the spreadsheet window. Again, use the Tracer to see what's going on in APL.

    NOTE: Control is not passed to APL until you take your fingers off both keys after invoking a macro with a short-cut key.

5. When you are comfortable with the workings of the system, interrupt the `⎕DQ` by typing Ctrl+Break or by selecting "Interrupt" from the *Action* menu in the Session Window.

    Execute `LX` (don't trace this time) and try invoking each macro by typing Ctrl+a (for Total) and Ctrl+b (for Mean) in the spreadsheet window. Use Ctrl+c to quit the application and APL.

6. Invoke APL again, and save the workspace EXCEL with a Latent Expression of `LX` i.e.:

    ```
    )LOAD EXCEL
    ⎕LX←'LX'
    )SAVE
    ```

    then exit from APL.

    In Excel, type Ctrl+a to calculate the Total. Excel tells you that Dyalog APL is NOT running, and asks if you want to start it. Reply YES. APL will start and load the EXCEL workspace. Since this now starts automatically, no Session Window will appear, although the functions are running. APL calculates the totals and sends them to Excel. Try successively typing Ctrl+a and Ctrl+b to calculate the totals and means.

We now have a hidden compute server for Excel. Type Ctrl+c to exit the application and APL.

Restrictions & Limitations

Although shared variables have been implemented as closely to the APL standard as is possible, certain restrictions are imposed by the nature of DDE itself.

1　　The server cannot make an offer to a specific client. Instead, it must broadcast a "general" offer, which could be accepted by any client. Indeed neither the client nor the server can specifically identify the other task.

2.　　Dyalog APL supports Excel "Fast Table Format" for communications with Excel (and with any other application that supports this format). This imposes the following restrictions :

The maximum number of numbers that you can send to Excel is 8191. Any attempt to send more will result in a LENGTH ERROR. This is because APL currently tries to send all the data in a single block. Larger amounts of data can be received from Excel, because Excel will send several blocks if required. The restriction may be lifted in due course.

The maximum length of a character vector (which represents a string within a cell) is 255.

3.　　A client APL program can only use indexed assignment to change the value of a shared variable if it already knows the up-to-date value of the variable, i.e. if its ⎕SVS is 0 0 1 1 or 1 0 1 0. An attempt to use indexed assignment on a variable whose ⎕SVS is 0 1 0 1 will cause a NONCE error.

4.　　Consider Excel as a server and APL as client with several warm links to an Excel spreadsheet e.g.:

```
'DDE:EXCEL|SHEET1' ⎕SVO 'X R1C1'
'DDE:EXCEL|SHEET1' ⎕SVO 'Y R2C2'
'DDE:EXCEL|SHEET1' ⎕SVO 'Z R3C3'
```

If R1C1 is changed in Excel, APL expects to be told only of that change. Instead, Excel tells APL that ALL the linked cells have changed.

If APL pokes a value back to R1C1, Excel again tells APL that ALL the linked cells have changed.

You must take care to avoid this problem when dealing with DDE between Excel and APL.

Index

DYALOG APL