MacApp® 2.0b5
Feature Overview

August 3, 1988
Curt Bianchi

This document is a brief description of the new features included in MacApp 2.0.

MacApp Source Code Organization

MacApp 2.0 breaks up the MacApp source code into logically separate units, each of which contains the code for a distinct part of MacApp. Here's a description of the new units.

Non–Object-Oriented Libraries

These libraries provide non–object-oriented support facilities for MacApp.

UMAUtil	This unit contains a set of constant and type declarations and utility routines that are used by the other MacApp units. MacApp users will also want to make use of the routines in this unit.
UViewCoords	This unit implements the routines dealing with 32-bit view coordinates and the `VPoint` and `VRect` data types, which are analogous to QuickDraw's `Point` and `Rect` types. MacApp users will have to be familiar with the routines in this unit. Basically, they provide conversion between QuickDraw and view coordinates, and view coordinate counterparts to many QuickDraw routines (for example, `SetVRect`, which is just like QuickDraw's `SetRect` except that it works in view coordinates).
UFailure	This unit implements MacApp's failure-handling mechanism. Most users only need to know how to invoke failure handling and how to write their own failure handlers. The code in this unit actually implements the failure-handling mechanism.
UPatch	This unit implements MacApp's trap-patching scheme and is of little interest to MacApp users as they don't ordinarily patch traps.
UBusyCursor	This unit implements the MacApp busy cursor. Most MacApp users won't be interested in this unit unless they want to change the way the busy cursor works.
UMemory	This unit implements the MacApp memory and segment management system. MacApp users need to know the philosophy behind MacApp's memory management and the services provided by this unit. If you need to know the gory details, read this unit's source code.
UMenuSetup	This unit implements MacApp's menu handling. Mostly this consists of routines to manipulate menu items via command numbers, and provides the framework in which menu setup takes place.

© 1988 Apple Computer

Object-Oriented Libraries

These units form the core of the MacApp object classes.

UObject	This unit provides the base support for objects in MacApp and includes the `TObject` class.
UAssociation	This unit implements the `TAssociation` class, which is essentially a dictionary that associates one string with another.
UList	This unit contains the `TList` and `TSortedList` classes, which implements lists of objects, similar to dynamic arrays. They are widely used by MacApp and by most MacApp programs.
UMacApp	This is the main MacApp unit. It contains the classes `TApplication`, `TDocument`, `TView`, `TWindow`, `TScroller`, `TControl`, `TCtlMgr`, `TScrollBar`, `TSScrollBar` and `TPrintHandler`. It also contains global variables, constants, and type declarations used by MacApp.

Building Block Units

These units implement optional classes you may wish to use in your program.

UTEView	This unit contains the `TTEView` class, which implements a text editing view based on the ROM TextEdit.
UDialog	This unit is essentially a collection of views that are generally associated with dialogs. The views defined by this unit are `TDialogView`, `TCluster`, `TPicture`, `TIcon`, `TRadio`, `TButton`, `TCheckBox`, `TStaticText`, `TEditText` and `TNumberText`. You can use these views in any MacApp window by including this unit in your USES statement.
UGridView	This unit contains a set of views for line- and column-oriented drawing. `TGridView` implements a spreadsheet-like view. `TTextGridView` implements a grid view whose cells are text-based. `TTextListView` is a view that consists of a list of text items.
UPrinting	This unit contains the `TStdPrintHandler` class, which allows you to print MacApp views.

Debugging Units

These units implement MacApp's debugging facilities.

UInspector	This unit contains the classes that implement the Inspector debugging windows.
UTrace	This unit implements the MacApp debugging facilities available in the Debug window.
UWriteLnWindow	This is the underlying implementation for the Debug window. It enables Pascal `WRITE` and `WRITELN`s to be written to the Debug window.

MPW 3.0 Support

We have made modifications to use MacApp 2.0b5 with MPW 3.0. To use MPW 3.0 you must change the `MacAppStartup` file in your MacApp folder to set the `MPW2` shell variable to false. MPW 3.0 is still changeing so we cannot guarentee that this release of MacApp will be compatible with future release of MPW 3.0.

At the time of this writing the current release of MPW 3.0 (version a2) contains incomplete `TextEdit` interfaces. These interfaces will be completed in the next release of MPW 3.0. Until then you will have to modify the source file `UTEView.p` to include the missing `TextEdit` interfaces. Near the beginning of `UTEView.p` you will find the following code:

```
{$IFC qMPW2}        { TextEdit interface that isn't included in MPW 2.x. }
CONST
  doToggle   = 32;

TYPE
  TEIntHook = (intEOLHook, intDrawHook, intWidthHook, intHitTestHook);

FUNCTION  TEContinuousStyle(VAR mode: INTEGER; VAR aStyle: TextStyle;
                            hTE: TEHandle): BOOLEAN;
                            INLINE $3F3C, $000A, $A83D;

PROCEDURE SetStylScrap(rangeStart, rangeEnd: LONGINT; newStyles: StScrpHandle;
                            redraw: BOOLEAN; hTE: TEHandle);
                            INLINE $3F3C, $000B, $A83D;

PROCEDURE TECustomHook(which: TEIntHook; VAR addr: ProcPtr; hTE: TEHandle);
                            INLINE $3F3C, $000C, $A83D;

FUNCTION  TENumStyles(rangeStart, rangeEnd: LONGINT; hTE: TEHandle): LONGINT;
                            INLINE $3F3C, $000D, $A83D;
{$ENDC qMPW2}
```

For MPW 3.0a2 this should be changed as noted in boldface:

```
{$IFC qMPW2}        { TextEdit interface that isn't included in MPW 2.x. }
CONST
  doToggle   = 32;

TYPE
  TEIntHook = (intEOLHook, intDrawHook, intWidthHook, intHitTestHook);
{$ENDC}

FUNCTION  TEContinuousStyle(VAR mode: INTEGER; VAR aStyle: TextStyle;
                            hTE: TEHandle): BOOLEAN;
                            INLINE $3F3C, $000A, $A83D;

PROCEDURE SetStylScrap(rangeStart, rangeEnd: LONGINT; newStyles: StScrpHandle;
                            redraw: BOOLEAN; hTE: TEHandle);
                            INLINE $3F3C, $000B, $A83D;

PROCEDURE TECustomHook(which: INTEGER; VAR addr: ProcPtr; hTE: TEHandle);
                            INLINE $3F3C, $000C, $A83D;

FUNCTION  TENumStyles(rangeStart, rangeEnd: LONGINT; hTE: TEHandle): LONGINT;
                            INLINE $3F3C, $000D, $A83D;
{Take out the $ENDC qMPW2 that was here}
```

View Architecture

A major part of the 2.0 effort is a reworking of MacApp's display objects. We have replaced the old `TWindow`, `TFrame` and `TView` objects with a new set of objects. The attached document "MacApp® 2.0 Display Architecture Release Notes" describes the changes in detail.

View Resource Templates

A major feature of MacApp 2.0 is the ability to create hierarchies of views from resource definitions in place of procedure calls. View resources are similar to a `'DITL'` resources and are used to describe MacApp views in much the same way as a `'DITL'` resources describe dialog items. View resources define the layout of views within a window. Nearly all fields of a view can be defined in its resource, including its size, location, adornment and text style, if any, and so on. MacApp provides a set of Rez type definitions so that the view resources can be compiled with Rez, or decompiled with Derez. Furthermore, a view resource editor is under development that allows you to create view resources interactively, much like ResEdit creates resources.

The use of view resources is purely optional. Instances of view classes provided by MacApp can be created procedurally or via resources. Within an application it is possible to have some views and windows created procedurally and have others created via resources. Which technique you should use is largely a matter of personal choice. However, a compelling reason for using resources is to allow your program to be localized (adapted to different natural languages) without affecting the source code.

Chapter Seven of the MacApp 2.*x* Manual provides a good introduction to using view resources. Also, many of the sample programs create their views and windows from resources. In particular, DemoDialogs has several examples and we would suggest that you look their for guidance.

In addition to the material in the manual, here are a few things you should know:

1. The MacApp 2.*x* Manual shows how to define 'view' resources in your application's .r resource file. With this technique the 'view' resources are described according to a syntax defined in the file ViewTypes.r in the 'MacApp Resource Files:' directory. This file includes the definition of the `'view'` type and shows what identifiers are permissible in each field. It would be helpful to print this file and have it on hand for reference as you create new `'view'` resources.

2. A single `'view'` resource consists of one or more views, much like a single `'DITL'` resource consists of one or more dialog items.

3. Each view in a `'view'` resource has a type and a class name. The type indicates to Rez and Derez the format of the rest of theview's data. The class name is the actual name of the view's class. If you were defining a TWindow object in a resource, its type would be `Window` and its class name would be `"TWindow"`. Similarly, if you were defining an object of class `TMyWindow`, then its type would still be `Window` but its class name would be `"TMyWindow"`. Note that the class name is case sensitive. Also, each type has a default class name assoicated with it. If the class name is an empty string then MacApp will use the default class. This decreases the size of the resource and provides better performance. For example, if you have a window view and you want its class to be `TWindow` then you can make the class name an empty string. On the other hand, if you have a window and whose class is `TMyWindow` then the class name must be `"TMyWindow"`.

4. The definition for the `'view'` resource is contained in the file 'ViewTypes.r' in the 'MacApp Resource Files:' directory. It is possible to extend the format of the `'view'` resource to include data for your own view classes. To do this you should make a copy of the `'view'` resource in one of your application's .r files and then add your types to the copied definition.

5. Note that each view can have a unique four-character identifier. By using the `TView.FindSubView` method you can get a reference to a view of a given identifier. This is useful when creating views from resources as you won't readily have references to any of the views in the hierarchy except the top one, usually a window object.

The four-character identifier does not have to be unique for views that you won't refer to by id.

6. For the creation of views with resources, the method `IRes` has been defined. It is analogous to its `IViewclass` method except that it initializes the class from resource data rather than from parameters and default values. Each `IRes` method is responsible for initializing the fields of the class to which it belongs. Generally, every view class has an `IRes` method. There are two reasons you may want to override `IRes` for your view classes. Either you wish to initialize some of your fields when the view is created from a resource, or you have added data to the resource specifically for this class. Here is an example of how `IRes` is implemented:

```
PROCEDURE TStaticText.IRes (itsDocument: TDocument;
                            itsSuperView: TView;
                            VAR itsParams: Ptr);

TYPE
   STextDataPtr =^STextData;
   STextData =   RECORD
                    just: INTEGER;
                    data: Str255;
                 END;

BEGIN
   fDataHandle := NIL;
   INHERITED IRes(itsSuperView, itsParams);

   fDefChoice := mStaticTextHit;
   WITH STextDataPtr(itsParams)^ DO
     BEGIN
     fJust := just;
     SetText(data, kDontRedraw);
     END;

   OffsetPtrWStr(itsParams, SIZEOF(STextData));
END;
```

The parameters `itsDocument` and `itsSuperView` are self-explanitory. The parameter `itsParams` points to the beginning of the view's resource data. It is a `VAR` parameter and each class's `IRes` increments `itsParams` by the length of the class's resource data. The standard `IRes` behavior is to 1) initialize any fields of the class required to free the object, 2) call `INHERITED IRes` to initialize data inherited from other classes, 3) initialize the class's data from the resource, and 4) increment `itsParams` by the size of the class's data. In the example shown above `itsParams` was offset with the utility `OffsetPtrWStr`, which is used when a variable length string is *at the end* of the class's data. Even though the string is declared as `Str255` in Pascal, in reality the number of bytes used by the string depends on its length. If there is no variable length string at the end of the data, then `OffsetPtr` is used. Given that each class's `IRes` method increments `itsParams` properly, `itsParams` will point to this class's portion of the resource after `INHERITED IRes` has been called.

7. Two `TEvtHandler` methods have been added to create views from resources. They are

```
FUNCTION TEvtHandler.DoCreateViews ( itsDocument: TDocument;
                                     parentView: TView;
                                     itsRsrcID: INTEGER): TView;

FUNCTION TEvtHandler.CreateAView ( itsDocument: TDocument;
                                   itsSuperView: TView;
                                   VAR itsParams: Ptr): TView;
```

The `DoCreateViews` method loads the `'view'` resource whose id is `itsRsrcID`, and then proceeds to

create the views defined in the resource. It calls `CreateAView` for each view in the resource. `CreateAView` basically clones a prototype view of the given class and then calls its `IRes` method. The global function, `NewTemplateWindow`, can be used to create a view hierarchy whose first view is a window. It is similar to `NewSimpleWindow` except that it creates a window and its subviews entirely from a resource and returns a reference to the window. (Actually it returns the first view in the resource under the assumption that it is a window.)

In order for `CreateAView` to clone a prototype view, an instance of every view that may be created from a resource must be included in a list of prototype views maintained by MacApp. To register a view, you use the following sequence:

```
VAR aMyView:
    TMyView;

NEW(aMyView);
FailNIL(aMyView);
RegisterType('TMyView', aMyView);
```

This sequence of code allows MacApp to create views of class `TMyView` as they are encountered in a `'view'` resource. Note that the string passed to register type is case-sensitive. Usually you register views in your `IYourApplication` method, although a view can be registered any time before you attempt to create it from a resource. All of the view classes defined in `UMacApp` are automatically registered. Calling `InitUTEView` registers `TTEView`. Similarly, `InitUDialog` initializes the views defined in `UDialog`. It is possible to use a different technique for creating instances of views by overriding `CreateAView`.

MultiFinder Support

While programs built with MacApp 1.*x* work under MultiFinder, they don't really take advantage of it.

MacApp 2.0 makes use of the `WaitNextEvent` trap and provides a mechanism for identifying the cursor region and length of time before MultiFinder needs to wake up the application. Three factors determine the value of the sleep parameter: the application's current "target," whether your application has any co-handlers, and the idle frequency of your application's event handlers. (*Event handlers* are objects whose classes descend from `TEvtHandler` and include the application itself, documents, windows and views.) Every event handler has a field, `fIdleFreq`, that determines how often the object requires idle processing. This field represents the minimum number of ticks (1/60 of a second) that must elapse before the event handler requires idle processing. By default, `fIdleFreq` is set to `MaxLongInt`, which for practical purposes means the object never requires idle processing. To cause an object to get idle time, simply set its `fIdleFreq` to an appropriate value. A value of zero will cause the object's `DoIdle` method to get called as often as possible. Note that there is no way to guarantee that the object's `DoIdle` method will be called after `fIdleFreq` ticks have elapsed. It is only guaranteed that its `DoIdle` method will not be called more frequently than the `fIdleFreq` ticks.

The application's current target determines which event handlers are eligible for idling. The target is the object that is the focal point of the user's interest. Usually it is a view in the front window. Event handlers form a linked list. Given that the target is a view of the front window, the *target chain* usually consists of the view, its window, its document, and the application. Thus all of those objects are eligible for idling. When another window is activated, a new view is made the target, and hence the target chain now consists of the new target, its window, its document, and the application. (Note that this technique is also used to determine which objects can respond to keystrokes and menu commands. See Chapter Five of the MacApp Interim Manual for more details.) The reason idle processing is usually restricted to objects in the current target chain is best explained by example. Consider a simple text editor such as the DemoText sample program. It is possible to open several windows, each with a text editing view. However, only the active window has a blinking edit caret, where the blinking of the caret is performed at idle time. If all of the views received idle time, there would be a blinking caret in all of the windows. The `TTEView` class is an example of an event handler that requires idling. In `TTEView`'s case, its `DoIdle` method is used to blink the

edit caret.

Sometimes it is desirable to have event handlers that receive idle processing regardless of the current target. These objects are known as *event co-handlers* in MacApp. They are installed in a list of co-handlers that is not affected by the current target. Possible uses of co-handlers would include objects that poll various I/O devices for input or output.

The cursor region passed to `WaitNextEvent` allows MultiFinder to avoid sending your application needless "mouse-moved" events. In MacApp 2.0, views can set the cursor region in their `DoSetCursor` method. `DoSetCursor` should set the region in view coordinates. MacApp will convert it to global coordinates as required by `WaitNextEvent`. Unless DoSetCursor is overridden the view return an empty cursor region, forcing MultiFinder to return mouse-moved events as long as the cursor is in the view.

TSortedList Class

MacApp 2.0 includes a subclass of `TList` called `TSortedList`, contained in the unit `UList`. `TSortedList` implements a list of objects that are maintained in sorted order. To do this there must be some way to rank objects in the list with respect to each other. For this reason the `TSortedList` class defines the `Compare` method. Its purpose is to compare two objects in a list and indicate which object is of greater rank. Since the implementation of `Compare` depends on the kind of objects maintained in the list, it must always be overridden. Its definition is:

```
FUNCTION  TSortedList.Compare (item1, item2: TObject): INTEGER;
```

`Compare` returns an integer that is less than, equal to, or greater than zero according to whether `item1` is less than, equal to, or greater than `item2`. As a convenience the following constants are defined:

```
kALessThanB =       -1;
kAEqualB =           0;
kAGreaterThanB =     1;
```

How you determine the result of `Compare` is completely up to you. As an example, suppose your objects have a field of type Str255 called `fTitle`, and you wish to maintain those objects in a list in ascending order according to `fTitle`. The `Compare` method could be implemented as follows:

```
FUNCTION  TSortedList.Compare (item1, item2: TObject): INTEGER;

BEGIN
  IF TMyObject(item1).fTitle < TMyObject(item2).fTitle THEN
    Compare := kALessThanB
  ELSE IF TMyObject(item1).fTitle > TMyObject(item2).fTitle THEN
    Compare := kAGreaterThanB
  ELSE
    Compare := kAEqualB;
END;
```

Note that `item1` and `item2` must be cast from type `TObject` to their actual type.

Other methods of the `TSortedList` class include:

`PROCEDURE TSortedList.Insert (item: TObject);`
Inserts the given object into the list in sorted order using the `Compare` method to compare objects.

`PROCEDURE TSortedList.GetItemNumber(item: TObject): INTEGER; OVERRIDE;`
Returns the item number within the list of the given object, using a binary search to locate the object within the list. Actually, the item number returned may not refer to the same object. The search is considered successful when the `Compare` method returns a result of zero. Depending on the kind of

objects in the list, your `Compare` method may return zero when comparing two entirely different objects. (e.g. it could be that more than one object has the same `fTitle` value in the above example.)

```
PROCEDURE TSortedList.Search (
                    FUNCTION TestItem(anItem: TObject): INTEGER): TObject;
```

This method can be used to perform a binary search on a sorted list, where the `TestItem` function is used to perform comparisons. `TestItem` returns an integer which is less than, equal to, or greater than zero according to whether `anItem` is less than, equal to , or greater than your search criteria. `Search` either returns the object that matches the search criteria, or `NIL` if no object matches.

`Search` is useful for cases in which you don't have an object to compare to those in the list. Refering to the example above, suppose we are given a string and wish to determine if any objects in a list have an `fTitle` that matches the string. This can be done with the following code fragment:

```
FUNCTION FindTitle (aTitle: Str255): TMyObject;

  FUNCTION TestTitle (anItem: TObject): INTEGER;
  BEGIN
    IF TMyObject(anItem).fTitle < aTitle THEN
      TestTitle := -1
    ELSE IF TMyObject(anItem).fTitle > aTitle THEN
      TestTitle := 1
    ELSE
      TestTitle := 0;
  END;

BEGIN
  FindTitle := gListOfMyObjects.Search(TestTitle);
END;
```

UDialog Building Block

Significant changes have been made for "dialog" support in MacApp 2.0. The old UDialog unit has been completely replaced. The thrust of the new dialog unit is to provide a set of view classes that implement the types of views commonly found in dialogs, and to do away with any reliance on the Dialog Manager.

The distinction between a dialog and a window in MacApp has been purposely blurred. In MacApp 2.0, windows and dialogs are constructed in basically the same way. Any view, including those in the UDialog unit, can be placed in any window. Any window can be modeless or modal. This is an improvement over MacApp 1.*x* in three respects. First, the techniques used to implement a dialog are the same as those used to implement any MacApp window. Second, it is easy to make use of controls in a nondialog context, since controls are implemented as MacApp views. And third, it is easier to have complex views in dialogs. The `TDialogView` class implements the basic behavior of modal dialogs (for example, tabbing between editable text fields, handling the default button, and so on).

The MacApp 2.0 UDialog unit is essentially a collection of view classes for the types of views one commonly uses in dialogs. Most of these views can be used in any MacApp window, in any context. The exceptions are the `TEditText` and `TNumberText` views. These views assume they are directly or indirectly installed in a TDialogView. The `TDialogView` handles tabbing among `TEditText` and `TNumberText` view. See the UDialog Release Notes for more information on the class provided in that unit.

Note that the notion of window modality is now a property of MacApp's `TWindow` class. `TWindow` has a field, `fIsModal`, that indicates whether the window is modal with respect to the other windows, regardless of its contents. A MacApp modal window still allows the menu bar to be clicked, but it doesn't allow other windows to

be activated. Note that MacApp modal windows do not by themselves prevent the application from being switched out by MultiFinder. This is done by MultiFinder, which prevents switching out if the front window's definition ID is `dBoxProc`.

One final note. MacApp does not necessarily preclude the use of the Dialog Manager. However, there is no support included for using the Dialog Manager.

UTEView Building Block

The UTEView building block implements the `TTEView` class. This is a view based on the features of the ROM's `TextEdit`. It supports styled text as defined in Inside Macintosh Volume V. The use of styled text requires system 6.0 or greater. The DemoText sample program provides an example of its use.

UGridView Building Block

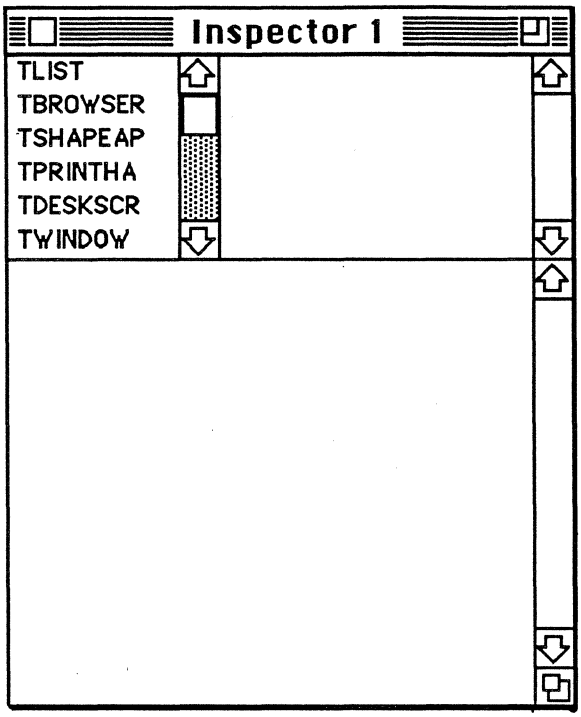
UGridView is a new building block in MacApp 2.0. It contains a set of view classes that are organized as rows and columns of cells, much like a spreadsheet. Examples of its use can be found in the samples DemoDialogs and Calc. The Inspector debugging windows also use UGridView. See the GridView Release Notes for more info.

Object Inspecting

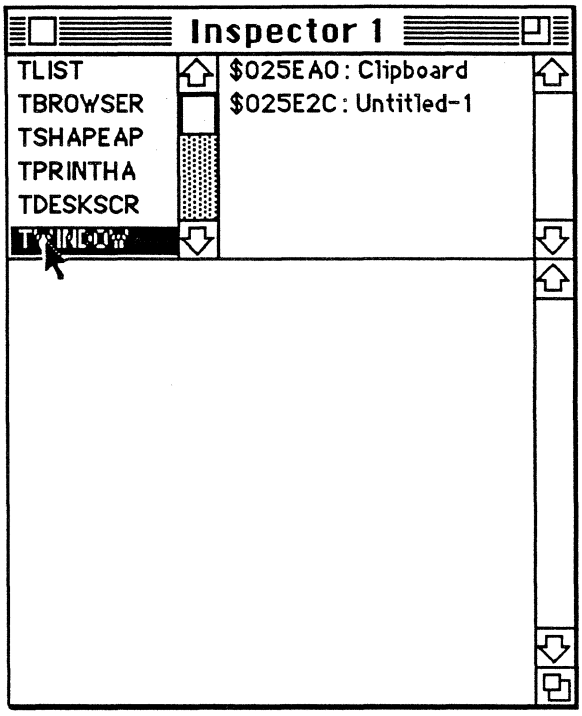
An object inspector has been added to MacApp's debugging facilities. The inspector provides an easy way to display the fields of any instantiated object. Each time the New Inspector Window command of the Debug menu is selected, a new inspector window is displayed. An inspector window is shown on the next page.

The upper left pane is a list of class names in alphabetical order for which at least one object has been instantiated. Clicking one of the class names fills the upper right pane with a list of objects of that class. Clicking one of the objects causes the bottom pane to display the fields of that object. Once an object is displayed in the bottom pane, it is also possible to click on a field that is one of the following types: a reference to another object, a `GrafPtr`, a `WindowPtr`, a `ControlHandle`, a `TEHandle`, or a `RgnHandle`. Clicking one of these fields causes the bottom pane to display the field that was clicked.

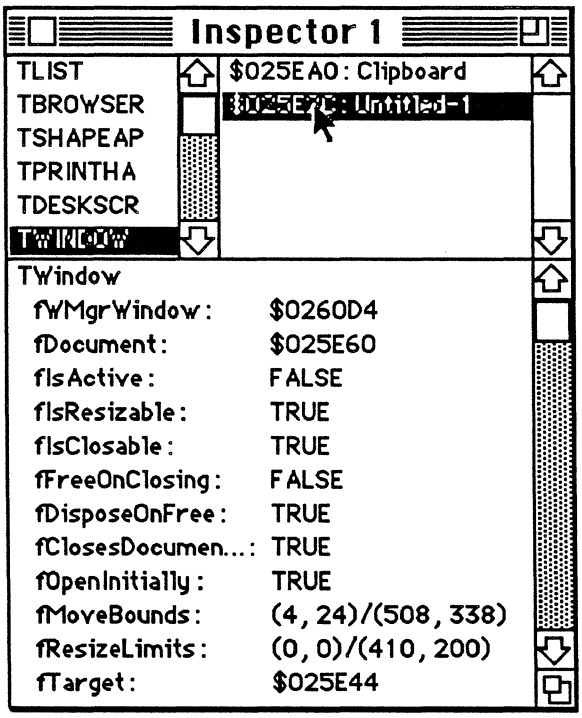
Note that the bottom pane is not automatically updated when changes occur in the data being displayed. (Refreshing the window, or clicking the object in the upper-right pane will reflect the object's current values.)



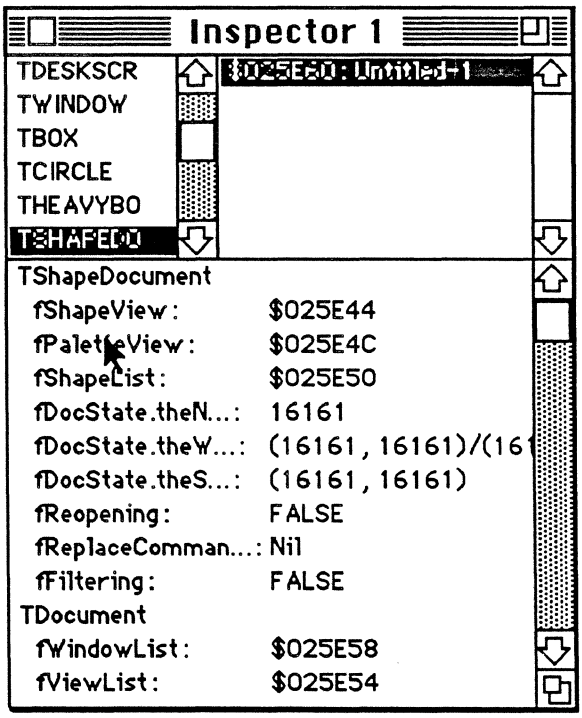
After selecting the New Inspector Window command from the Debug menu



After clicking on "TWindow" in the list of classes



After clicking on the window 'Untitled-1'



After clicking on the fDocument field of the window

© 1988 Apple Computer

To implement the inspector, we've added two new methods to objects: `TObject.GetInspectorName` and `TObject.Fields`. (MacApp 1.x users should note that the `Inspect` method is still around, and will still work in the Debug Window. To make use of the Inspector windows you should replace your `Inspect` methods with `Fields` methods.)

`GetInspectorName` is used to provide some identification of the object when it is listed in the upper right pane of an Inspector window. For example, the method `TWindow.GetInspectorName` returns the window's title. It is defined as:

```
PROCEDURE TObject.GetInspectorName (VAR inspectorName: Str255);
```

The `Fields` method returns information about the fields of a particular class. It has already been implemented for all classes defined by MacApp, so you only need to implement it for your classes. The definition of `Fields` is:

```
PROCEDURE TObject.Fields (PROCEDURE DoToField( fieldName: Str255;
                                               fieldAddr: Ptr;
                                               fieldType: INTEGER ));
```

The purpose of the `Fields` method is to call `DoToField` on each field defined by its class. It is used by the Inspector windows as well as for the Inspect command in the MacApp debugger. The general sequence of a `Fields` method is to first call `DoToField` to report the class name, then call `DoToField` for each field in the class, and finally call `INHERITED Fields(DoToField)` for the inherited data. For example, suppose we have the class defined below:

```
TShape =   OBJECT (TObject)
               fRect:  Rect;
               fColor: RGBColor;
           END;
```

The `Fields` method for `TShape` would be:

```
PROCEDURE TShape.Fields (PROCEDURE DoToField( fieldName: Str255;
                                              fieldAddr: Ptr;
                                              fieldType: INTEGER ));
                                              OVERRIDE;

BEGIN
   DoToField('TShape', NIL, bClass);
   DoToField('fRect', @fRect, bRect);
   DoToField('fColor', @fColor, bRGBColor);
   INHERITED Fields(DoToField);
END;
```

The constants bClass, bRect, and bRGBColor are among constants defined in UMAUtil and indicate the type of data pointed to by the second parameter to DoToField. Here is the complete list of field type constants defined in UMAUtil:

bInteger	= -1;	an INTEGER, converted to a decimal string.
bHexInteger	= -2;	an INTEGER, converted to a hexdecimal string.
bLongInt	= -3;	a LONGINT, converted to a decimal string.
bHexLongInt	= -4;	a LONGINT, converted to a hexidecimal string.
bString	= -5;	a STRING of any length.
bBoolean	= -6;	a BOOLEAN.
bChar	= -7;	a CHAR.
bPointer	= -8;	a pointer of any type, converted to a hexidecimal string.
bHandle	= -9;	a handle of any type, converted to a hexidecimal string.
bPoint	= -10;	a QuickDraw Point.
bRect	= -11;	a QuickDraw Rect.
bObject	= -12;	a reference to another object.
bByte	= -13;	a single byte, converted to a decimal string.
bCmdNumber	= -14;	a field of type CmdNumber.
bClass	= -15;	announces the start of a class's fields.
bOSType	= -16;	a field of type OSType, converted to a 4-character string.
bWindowPtr	= -17;	a WindowPtr.
bControlHandle	= -18;	a ControlHandle.
bTEHandle	= -19;	a TEHandle.
bLowByte	= -20;	the low byte of a 2-byte word (integer), converted to a hex string.
bHighByte	= -21;	the high byte of a 2-byte word (integer), converted to a hex string.
bPattern	= -22;	a QuickDraw Pattern.
bFixed	= -23;	a field of type Fixed.
bRgnHandle	= -24;	a RgnHandle.
bRGBColor	= -25;	a field of type RGBColor.
bTitle	= -26;	field data is ignored–only its title is displayed in an Inspector window.
bGrafPtr	= -27;	a pointer to a QuickDraw graf port.
bStyle	= -28;	a QuickDraw Style record.
bVCoordinate	= -29;	a VCoordinate.
bVPoint	= -30;	a VPoint.
bVRect	= -31;	a VRect.
bFontName	= -32;	a font number, converted to a font name.
bStringHandle	= -33;	a handle to a string.
bCntlAdornment	= -34;	a field of type CntlAdornment.
bIDType	= -36;	a field of type IDType, converted to a 4-character string.
bResType	= -37;	a field of type ResType, converted to a 4-character string.

Record structures can be inspected by calling DoToFields for each field in the record. The procedure TextStyleFields in UMAUtil is an example of this. It is defined as:

```
PROCEDURE TextStyleFields (aTitle: Str255; VAR aStyle: TextStyle;
                    PROCEDURE DoToField (fieldName: Str255;
                                         fieldAddr: Ptr;
                                         fieldType: INTEGER));
BEGIN
  DoToField(aTitle, NIL, bTitle);
  DoToField('  Font', @aStyle.tsFont, bFontName);
  DoToField('  Face', @aStyle.tsFace, bStyle);
  DoToField('  Size', @aStyle.tsSize, bInteger);
  DoToField('  Color', @aStyle.tsColor, bRGBColor);
END;
```

Note that the TextStyle record passed to TextStyleFields *must* be a VAR parameter because each call to

DoToField in TextStyleFields passes an address within the TextStyle record–these addresses are converted to offsets within the object being inspected unless the object is the application object. If the TextStyle record wasn't a VAR parameter then you would be passing addesses within a copy of the TextStyle record on the stack. Note that if the object is the application object then the address returned by DoToField is considered an address to global data. Thus you can inspect your application's global data by overriding the Fields method for your application object.

It is also possible to extend the types of fields that can be inspected to include your own data types. Here's what to do to add your own field types:

1. Define constants for the field types that you handle. By convention the names of these constants start with a 'b'. MacApp reserves the numbers 0 to MAXINT and -1 to -99 for its use. Applications can use numbers less than -99.

2. Define a procedure with the following interface:

```
PROCEDURE MyFieldToString (theData: Ptr;
                           fieldType: INTEGER;
                           VAR theString: Str255);
```

   The purpose of this routine is to convert the data pointed to by theData, and whose type is defined by fieldType, into the string theString. UMAUtil has already implemented a routine such as this called StdFieldToString, which converts data of any type described above (from bInteger to bResType) into a string. Your routine should use a CASE statement to convert the data types that it is capable of converting, and call StdFieldToString if it doesn't handle the given data type.

3. In your application object's initialization method set the global variable gFieldToStrRtn to point to your field-to-string routine.

An example implementation is shown on the next page. It converts floating-point data types to string for the purpose of inspecting floating-point fields in objects.

```
CONST
  bReal =                   -100;
  bSingle =                 -101;
  bDouble =                 -102;
  bExtended =               -103;

PROCEDURE TMyApplication.IMyApplication;
BEGIN
  ...
  gFieldToStrRgn := @MyFieldToString;
  ...
END;

{$IFC qDebug}
{$IFC qTrace}{$D+}{$ENDC}
{$S MADebug}
PROCEDURE MyFieldToString (theData: Ptr;
                           fieldType: INTEGER;
                           VAR theString: Str255);

TYPE
  TAlias =      RECORD
                  CASE INTEGER OF
                    bReal,
                    bSingle:            (asReal: REAL);
                    bDouble:            (asDouble: DOUBLE);
                    bExtended:          (asExtended: EXTENDED);
                    END;
VAR
  alias:        ^TAlias;
  aDecForm:     DecForm;
  x:            EXTENDED;

BEGIN

  { Note this hasn't been compiled and is only an illustration of how
    to implement a routine of this type.  It may have errors. }

  alias := Pointer(theData);
  WITH alias^ DO
    CASE fieldType OF
      bReal,
      bSingle:
        BEGIN
        aDecForm.style := FloatDecimal; aDecForm.digits := 2;
        x := asReal;
        NumToStr(aDecForm, x, theString);
        END;
      bDouble:
        BEGIN
        aDecForm.style := FloatDecimal; aDecForm.digits := 2;
        x := asDouble;
        NumToStr(aDecForm, x, theString);
        END;
      bExtended:
        BEGIN
        aDecForm.style := FloatDecimal; aDecForm.digits := 2;
        NumToStr(aDecForm, asExtended, theString);
        END;
      OTHERWISE
        StdFieldToString(theData, fieldType, theString);
      END;
END;
{$IFC qTrace}{$D++}{$ENDC}
{$ENDC qDebug}
```

The Debug Window Resource

Various attributes of the Debug Window can now be defined in your application's resource file by including a resource of type `'DBUG'` and id 300. The format of the `'DBUG'` resource is:

```
type 'dbug' {
            rect;                       /* Bounding rect for debug window */
            integer normal = 4;         /* Debug window font rsrc ID */
            integer normal = 9;         /* Debug window font size */
            integer normal = 25;        /* Number of lines */
            integer normal = 80;        /* Width of lines in characters */
            pstring;                    /* Window title */
            };
```

The rect defines the window's bounding rectangle in global coordinates. The first two integers define the font and font size of the text in the Debug Window. `normal` refers to Monaco-9. The last two integers define the number of lines of text to retain in memory for scrolling, and the number of characters per line. MacApp will allocate a buffer whose size is the number of lines * the characters per line.

A List of MacApp 2.0 View Classes

One of the major efforts of MacApp 2.0 is to provide a richer set of views from which to work. To this end, the following view classes have been implemented:

UMacApp View Classes

`TView`	An *abstract class* (one which must be overridden in order to produce something useful) that defines a set of features and operations common to all views. These features include nesting, drawing, mouse handling, moving, and resizing.
`TWindow`	A subclass of `TView` that implements a Toolbox window. The window may be modal or modeless.
`TScroller`	A subclass of `TView` that is able to "scroll" its contents by effecting a coordinate transformation.
`TControl`	A subclass of `TView`, `TControl` is an abstract class that defines the features and operations common to all MacApp controls, of which the Control Manager controls are considered a subset.
`TCtlMgr`	A subclass of `TControl`, `TCtlMgr` is an abstract class that implements Control Manager controls.
`TScrollBar`	A subclass of `TCtlMgr` that implements scroll bars generically.
`TSScrollBar`	A subclass of `TScrollBar` that implements scroll bars specifically for scrolling MacApp views.

UTEView View Classes

`TTEView`	A subclass of `TView` that implements a text edit view based on the ROM TextEdit.

UDialog View Classes

`TButton`	A subclass of `TCtlMgr` that implements push buttons.
`TRadio`	A subclass of `TCtlMgr` that implements radio buttons.
`TCheckBox`	A subclass of `TCtlMgr` that implements check boxes.
`TDialogView`	A subclass of `TView` that implements a "dialog." A dialog view is used to provide tabbing between editable text fields, and to implement some standard behavior for the Return key and for push buttons.
`TCluster`	A subclass of `TControl` that is used to organize a set of views as a single group for the purpose of localizing relationships between the views. The most typical use of a cluster is to group together a set of radio buttons.
`TPicture`	A subclass of `TControl` that displays a QuickDraw picture.
`TIcon`	A subclass of `TControl` that displays an icon.
`TPattern`	A subclass of `TControl` that fills itself with a pattern.
`TStaticText`	A subclass of `TControl` that displays static (uneditable) text.
`TEditText`	A subclass of `TStaticText` that displays text and allows it to be edited. Dialog views implement tabbing between edit text views.
TNumberText	A subclass of `TEditText` that displays an integer value and allows it to be edited.

UGridView Classes

`TGridView`	A subclass of `TView` that implements a line- and column-oriented view such as a spreadsheet.
`TTextGridView`	A subclass of `TGridView` specifically for text.
`TTextListView`	A subclass of `TTextGridView` that implements a single-column list of text items, such as the list of files in a Standard File dialog.

The Samples

Here is a list of the sample programs included with MacApp 2.0.

Calc	A conversion of the spreadsheet program that appeared on one the MacApp Developers Association disks. It demonstrates the use of the GridView building block. It should be considered a "work–in–progress" as it is still somewhat buggy and feature incomplete.
Cards	A conversion of the MacApp 1.*x* program. It implements a simple note card file and a disk-based document object.
DemoDialogs	A completely new program, bearing no relation to the 1.*x* DemoDialogs. It demonstrates the use of the dialog building block and defining views in resources. It

	shows how modal dialogs can be used and implements a couple of custom controls.
DemoText	A conversion of the 1.*x* program that shows multiple styles in a single `TEView` and the use of view resource templates.
DrawShapes	A conversion of the the 1.*x* program.
Nothing	The simplest MacApp program, converted to 2.0. It also demonstrates the use of view resources rather than creating views procedurally.
PatView	A program that allows views to be drawn and moved about a background view, much as DrawShapes does with shapes. This program demonstrates the use of a large view (a view greater than QuickDraw's coordinate space) and view nesting.
Puzzle	A conversion of the MacApp 1.*x* program.
TwoDocKinds	A conversion of the MacApp 1.*x* program.