

# MacApp® 2.0b5

August 3, 1988

Welcome to MacApp 2.0b5. Much work has gone into MacApp since version 1.1.1 and some parts of MacApp 2.0 differ significantly from its predecessor. These release notes describe the major changes in MacApp.

The software in this package is considered beta quality. It has known bugs, and we're sure users of MacApp will find even more bugs. While we are not preventing you from shipping products based on this release of MacApp, please bear in mind that you do so at your own risk, with the understanding that this release of MacApp is not considered production quality. We encourage you to report bugs in the software or documentation, as well as suggesting improvements, by using the MacApp® Bug Report Form included in this package.

This release of MacApp is designed to work as is with MPW 2.0.2. It is also intended to work with the upcoming release of MPW 3.0 with little change. See the MacApp® 2.0b5 Feature Overview for more about using MPW 3.0. And please note that at the time of this writing MPW 3.0 is still changing so we cannot absolutely guarantee compatibility with the MPW 3.0 release that eventually is made available from APDA.

At this time the MacApp 2.0 Manual has not been completed. Even with these release notes and the MacApp® 2.x Manual (interim version) there are major portions of MacApp that are not documented. However, most of the differences between 1.x and 2.0 are documented in these release notes.

This package contains the following documents:

- An important letter from Apple's Software Licensing Department regarding distribution of MacApp and applications built with it.
- A copy of the MacApp® Object Code Distribution Agreement.
- MacApp® Bug Report Form, which contains the information necessary to report any software or documentation bugs you may encounter in MacApp.
- MacApp® 2.0b5 Known Bugs and Problems, which describes the known bugs and problems in this release of MacApp.
- MacApp® 2.0b5 Delta List, which describes the differences between this release and MacApp 2.0b2.
- MacApp® 2.0b5 Feature Overview, which describes the major new features of MacApp 2.0.
- MacApp® 2.0b5 View Architecture Release Notes, which documents the MacApp 2.0 view classes.
- MacApp® 2.0b5 UTEView Release Notes, which describes the contents of the UTEView unit and the TTEView class.
- MacApp® 2.0b5 UDialog Release Notes, which describes the contents of the UDialog unit and the various view classes contained therein.
- MacApp® 2.0b5 UGridView Release Notes, which describes the contents of the UGridView unit and its classes.
- MacApp® 2.0b5 Printing Release Notes, which describes how printing works in MacApp.

- **MacApp® 2.0 Memory Management**, which discusses memory management in MacApp.
- **MacApp® 2.0 Globals**, which describes global constants, variables, data types, and routines in MacApp.
- **MacApp® 2.0 Object and Method Reference**, which discusses object classes that also existed in MacApp 1.1.

To use MacApp we strongly recommend that you also have these products:

- **MacApp® 2.x Manual (interim version)**, available from APDA. This is essential for all MacApp users. It contains an introduction to object-oriented programming and Object Pascal, an overview of MacApp's organization and architecture, a rudimentary cookbook with programming examples, and a description of MacApp debugging tools.
- **Inside Macintosh Volumes I-V**, Apple Computer, Inc., Addison-Wesley Publishing Company. These books are essential for any Macintosh program, even with MacApp. They are available from APDA and many bookstores.
- **MacApp® 2.0b5 Source Listings**, available from APDA. This package consists of cross-referenced source code listings of MacApp 2.0b5.

Additional software and literature that should prove useful to anyone using MacApp includes the following products:

- **Object-Oriented Programming for the Macintosh**, Kurt Schmucker, Hayden Book Company. Available from APDA and in many bookstores. The description of MacApp is based on MacApp 1.x. However the description of object-oriented programming is still very applicable.
- **Object-Oriented Programming: An evolutionary Approach**, Brad Cox, Addison-Wesley Publishing Company. Available from APDA and in many bookstores. A good introduction to object-oriented programming.
- **Programming with Macintosh Programmer's Workshop**, Joel West, Bantam Books. Available from APDA and in many bookstores. A thorough description of MPW and an introduction to MacApp 1.x.
- **MacApp Developer's Association Newsletter**, a regular publication filled with information of interest to MacApp users. A six issue subscription is available directly from the Association for \$20 (or \$30 outside the U.S.) MacApp Developer's Association, P.O. Box 23, Everett, WA, 98206-0023.
- **MacApp Browser**, a handy desk accessory that makes it easy to read through Object Pascal source code (including MacApp and your own MacApp applications). Available from the MacApp Developer's Association and APDA.
- **MacApp Object Libraries 1-8**, consisting of member contributed software based on MacApp 1.x. Available from the MacApp Developer's Association.

## Using MacApp 2.0b5 With MPW 3.0

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 and set the MPW2 shell variable to false. MPW 3.0 is still changing so we cannot guarantee 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 MacApp 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}
```



## MacApp 2.0b5 Known Bugs

This section describes known bugs and problems in the 2.0b5 release.

### General Problems

1. MacApp is not yet as robust as we would like in low memory conditions.
2. MacApp has not been thoroughly tested with heap scrambling.
3. The samples tend not to be robust in low memory conditions.
4. The Pascal compile-time variables qTemplateViews, qProceduralViews and qWriteTemplates are not completely implemented and shouldn't be set to false.
5. Applications built with this release of MacApp will not run on a Macintosh XL (Lisa).

### UDialog

1. Undo for TEditText and TNumberText is disabled because it had bugs that could cause the program to crash. It will be reinstated in a future release.
2. Invalid character entry in a TNumberText view produces a beep and not an alert. Since the text typed into a TNumberText view is not validated until the user tabs to another view or attempts to close the window, it is possible that the application will prevent the user from quitting if he has typed invalid text into a TNumberText view and the view is still selected. This behavior can be modified by overriding TNumberText.Validate or TDialogView.CantDeselect.

### UGridView

1. Dim hiliting has been disabled because of bugs. It will be reinstated in a future release.
2. Resizing columns or rows causes the selected cells to be deselected.
3. FirstSelectedCell and LastCellSelected are not guaranteed to work unless the selection is a rectangle.

### UInspector

1. Using the Inspector in low memory situations may crash the program.
2. The Inspector sometimes gets confused about which items are selected in the upper two panes. For example, if the class TList is selected and another TList object is created then the Inspector may get confused.

### UList

1. The Delete operation in a TSortedList object does not use a binary search to find the object to be delete. Instead, it uses the same sequential search as implemented for TList. This will be corrected in a future release.

### UMacApp.TCommand

1. There is a fundamental problem with command objects in that they don't have a notion of a target on which the command operates. This becomes a problem for undoable commands whose document is NIL. In that case MacApp assumes the command is an application command and will allow the user to undo it even if the context no longer makes since for the command to be undone. For example, suppose a command operates on a document-less view. Even if the view is freed the command will still be undoable until another undoable command is created.

### UMacApp.TView

1. Shadow adornments do not print correctly.

### UPrinting

1. It is possible to cause a crash by undoing the Page Setup command if the view associated with the command has been freed. This situation will most likely occur if a document has more than one window, the Page Setup command is used while one of the windows is active, then the window is closed.
2. The `fEffectiveDeviceRes` field of `TStdPrintHandler` is not always accurate. We suggest that you avoid relying on this field as we intend to take it out completely in a future release.

### UTView

1. The interface for `TECustomHook` is incorrect. The first parameter should be an `INTEGER` rather than the enumeration `TEIntHook`. The enumerated identifiers in `TEIntHook` should be made constants. This needs to be changed only if you use `TECustomHook`. It is not used by `MacApp`.
2. There is some flashing of characters as lines of text are added to a `TEView` (i.e. parts of characters on the line being added are displayed twice). This should be fixed in a future release.

### UViewCoords

1. `Pt2VRect` does not work correctly if the horizontal or vertical coordinate of the first point is greater than the horizontal or vertical coordinate of the second point. It does work correctly if both coordinates of the first point are less than both coordinates of the second point, or both coordinates of the first point are greater than both coordinates of the second point. This routine is not used by `MacApp`, so it is not affected.
2. In the file `UViewCoords.p` we inadvertently left in a Pascal compiler include directive for `UViewCoords.inc1.p`, even though it doesn't exist. This generally will not cause problems as the `UViewCoords.p` unit is not compiled by `MABuild`. (The actual code for the routines defined in the unit is in assembly language.)

## **Changes Since MacApp 2.0b2**

This document lists the differences between `MacApp 2.0b5` and `MacApp 2.0b2`.

### CommonObjLib.a

1. Fixed bug in which the non-optimized method table cache fragmented the heap.
2. The non-optimized method table cache now does a Macsbug break when there is a nil object or method. It used to do a division by zero to cause a break.

### ObjLib.inc1.p

1. Fixed bug in which the non-optimized method table cache fragmented the heap.

### UAssociation

1. Changed some segment mappings.

## UBusyCursor

1. Changed name of watchCursor field in UBusyCursor.a to avoid conflict with MPW 3.0 interfaces.
2. Changed some segment mappings.

## UDialog

1. TDialogView.DoSelectEditText and SelectEditText always select the characters when the selectChars parameter is true. Previously it didn't select the characters if the edittext view was the one being edited.
2. Fixed bugs in TStaticText and TEditText.SetText in which the text didn't get redrawn if it was set to an empty string.
3. Modified TDialogTEView.InstallEditText so that it sets its height to be at least as high as a single line of text.
4. Change some segment mappings.
5. Fixed problems with nested clusters and radio buttons.
6. Added the methods TRadio.SetState and TCheckBox.SetState.
7. Added TDialogView.DoCommandKey to handle the command-'.' key combination.
8. The default control is highlighted if the users presses return. Similarly, the cancel control is highlighted if the user presses command-'.'.
9. Filled out the Fields methods.
10. Fixed bug in which compilation of TDialogTEView.Fields wasn't conditional on qDebug.
11. Fixed bugs in TStaticText and TEditText.Draw in which they weren't disposing of handles they created.
12. Added TStaticText and TEditText.DoSubstitution.
13. Added TStaticText and TEditText.ImageText.
14. TPopup now works correctly with respect to color.
15. Added the global routine GetMenuColors, which returns the correct color values for any menu item or menu title.
16. Added TPopup methods CalcLabelRect, DrawLabel and DrawPopupBox.
17. Fixed other TPopup oddities.
18. TPopup.SetPopup now calls MAInsertMenu instead of the Toolbox routine InsertMenu to make sure the menu's color table gets installed.
19. Fixed bugs in TPopup.IPopup and TPopup.IRes in which the menu and resource id fields were not set if the resource id was kNoResource. Also, these routines make the popup invisible if the machine does not support popup menus.
20. Now check for presence of 128K ROMs before using TESelView and TESysJust.
21. Modified TDialogView.PoseModally to set fDismissed to false before handling events. This fixes a bug when calling PoseModally more than once on the same instantiation of a TDialogView.
22. Fixed bug in which a range check error could occur when doing string substitutions on static text strings.
23. Modified TNumberText.Validate so that it considers an empty string to be the number zero. Previously it considered an empty string to be invalid.
24. Removed the \$B- compiler directive as it is not necessary.
25. Disabled undo in TEditText and TNumberText views because it had bugs that could cause the program to crash.

## UGridView

1. Some code has been added to handle dim highlighting. However, it is not yet debugged so it has been disabled.
2. Added the method DoHilite to handle cell highlighting.
3. Fixed bug in InvalidateSelection in which it invalidated an extra row and column.
4. Removed InitUGridView. It used to register TGridView, TTextGridView and TTextListView,

- but since these must be overridden to be useful, there is no point in registering them.
5. The SetPen method now sets the pen according to fTextStyle rather than taking a TextStyle parameter.
  6. Fixed bug in TGridView.Free in which some handles weren't being freed.
  7. SetRowHeight and SetColWidth now do nothing when attempting to set the height/width of a row/column to its existing height/width.
  8. Added the DrawRangeOfCells method which is used to draw a set of cells from a top-left to bottom-right cell.
  9. Added the methods FirstSelectCell and LastSelectedCell. Note that these are not guaranteed to work if the selection is non-rectangular.
  10. Changed VPointToCell to return its result as a VAR parameter rather than as a function result.
  11. Fixed bug in SetRowHeight in which the height was occasionally set incorrectly.
  12. Fixed bug in shift-key selection in which the "anchor" cell was incorrect.
  13. Fixed off-by-one bugs when redrawing.
  14. Fixed various bugs related to selection highlighting.
  15. Deleted the method HiliteCell.
  16. Changed kDoHighlight to kHighlight.
  17. Fixed bugs in DelRowLast and DelColLast when deleting multiple rows or columns.

### UInspector

1. The Inspector views are created via templates if the compile-time variable qTemplateViews is true.
2. Now avoid inspecting an object if its address is invalid (i.e. odd).
3. Fixed bug in which inspecting the fClassesByID and fClassesByName fields of the Inspector document caused a crash.

### UList

1. Changed segment mappings for TSortedList to be consistent with TList.
2. Removed TSortedList.Delete as it didn't work properly. (TList.Delete is used instead.)
3. For TSortedList.Compare and Search, added the constants kALessThanB, kAEqualB, kAGreaterThanB.

### UMacApp

1. Made all global variables part of the MacApp interface.
2. Added FindStdView and RegisterStdView routines to support "standard" view types.
3. Avoid registering view types if the compile-time variable qTemplateViews is false.
4. Fixed bugs in GetFocus/SetFocus in which they did not save/restore gLongOffset.
5. Fixed bug in which the low memory global SysFontSize was accessed on 64K ROMs, where it wasn't maintained. This produced a bad font size in the gSystemStyle record.
6. Modified MacAppAlert to set gCursorRgn to an empty region to ensure that the cursor gets reset after the alert.
7. Now make use of compile-time variables qProceduralViews, qTemplateViews and qWriteTemplates.
8. Removed some old TFrame-related constants.
9. Renamed 'isVisible' parameters to 'redraw' for consistency.

### UMacApp.TApplication

1. Fixed bug in OpenDeskAccessory in which an address error could occur.
2. The Clipboard window is now created via templates if the compile-time variable qTemplateViews is true.
3. Fixed bug in which gIdlePhase did not get set to gIdleBegin, thereby preventing

- UloadAllSegments from being called.
4. TrackMouse now calls the command object's AutoScroll method to carry out autoscrolling rather than handling the scroll directly.
  5. Modified GetEvent so that it doesn't call Idle on mouse-moved events if there are other events pending. (This is because MultiFinder effectly assigns a higher priority to mouse-moved events than other events.)
  6. Modified GetEvent so that it calls Idle before calling WaitNextEvent if there are no events pending. This ensures that the menu bar and cursor are properly set.
  7. No longer need to fake up activate events for desk accessories in HandleSystemEvent.
  8. Updated the Fields method.
  9. Changed the CommandKey method to DoCommandKey.
  10. Moved non-target related menu setup code from DoSetupMenus to SetupTheMenus so that the code gets executed even if the target chain doesn't include the application object. This is mostly debugging code and enabling/disabling the Apple menu.
  11. IApplication no longer looks for an 'mctb' resource whose id is that of the 'mbar'. Instead, you should use 'mctb' resources whose ids are the same as the menus you are installing and let the Menu Manager load the 'mctb' for you.
  12. In GetRsrcWindow, the result of GetNewCWindow is cast to a WindowPtr.
  13. Fixed bug in which the scrap didn't get exported when switching out to another application. The problem was that gClipWrittenToDeskScrap was set to false in SetClipView, when it shouldn't have been set to false if the clip view was gClipOrphanage.
  14. In SetTarget, gCursorRgn is set to an empty region to ensure that the cursor gets reset.
  15. Modified the failure handler in PollEvent to call HiliteMenu(0) to ensure that the highlighted menu title, if any, gets unhighlighted.

#### UMacApp.TCommand

1. Added the AutoScroll method, which can be overridden to change the way the command handles autoscrolling.
2. Added the document parameter to ICommand, making it explicit which document the command effects. Previously ICommand simply set fDocument to gDocument.

#### UMacApp.TControls

1. Fixed bug in which scroll bars were initially created as being active, whereas they are now initialized as inactive and made active when the Activate method is called. This bug manifested itself when opening more than one window at a time.
2. Fixed bug in which Control Manager controls were not dimmed when created from a resource which set the dim state to true.
3. Changed some segment mappings.
4. Added TControl.Resize to redraw the entire control when resized. Previously it only redrew the difference between the old and new size.
5. Fixed bug in TCtrlMgr.DimState in which it ignored the redraw parameter.
6. Modified TCtrlMgr.Hilite so that turning on hiliting sets the fCtrlMgrControl's hilite to 10. This makes the control look as though the mouse is clicked in it.

#### UMacApp.TDeskScrapView

1. PICT data is now shown in preference to TEXT if both TEXT and PICT data are in the desk scrap.

#### UMacApp.TDocument

1. Removed the method GetPriviledges and the field fAccessMode. (The notion of access modes

is not being supported.)

2. All calls to OpenFile are now routed through the method OpenAFile.

### UMacApp.TEvtHandler

1. CreateAView now calls FindStdView if the class name in the view's template is an empty string.
2. Added the DoCommandKey method so that any event handler can handle command-key events.
3. In CreateAView, added FailNIL after ShallowClone to catch failures in ShallowClone.
4. Added failure handling to DoCreateViews so that it frees the view hierarchy it's creating if a failure occurs along the way.

### UMacApp.TScroller

1. Fixed bug in AdjustScrollBars in which the scroll bars didn't get redrawn properly if they changed size but didn't move.
2. AutoScroll no longer checks for the presence of scroll bars.
3. DoScroll now resets the value of gLongOffset if the scroller is the focused view. This fixes bugs in which gLongOffset might be inaccurate after scrolling.
4. DoScroll now makes sure the deltas are within range before proceeding.
5. Renamed 'isVisible' parameters to 'redraw' for consistency.

### UMacApp.TView

1. Fixed bug in Resize in which it sometimes computed the frame size incorrectly.
2. Changed some segment mappings.
3. Added the redraw parameter to Show, and logic to redraw the view if the redraw parameter is true.
4. Modified DrawContents to improve performance.
5. AddSubview now sets the subview's fNextHandler field to the superview if fNextHandler is nil. Similarly, RemoveSubview now sets the subview's fNextHandler to nil if the subview's fNextHandler is the superview.
6. Removed some of the gltenseDebugging messages.
7. ContainsPoint and DrawContents now check IsShown.
8. DrawContents passes the correct rectangle to Draw. (It may not have, depending on the previous version of MacApp you had.)
9. Changed Activate to use fHLDesired as the fromHL parameter to DoHighlightSelection. This fixed bugs in which the highlighting might have been incorrect when first displaying the window.

### UMacApp.TWindow

1. Modified HandleMouseDown so that clicking an inactive window whose fDoFirstClick field is true causes the window to be selected *and activated* before handling the click.
2. SetResizeLimits now also sets the zoom size limit.
3. IRes now casts the result of NewCWindow into a WindowPtr.
4. Modified Resize to call SizeWindow without invalidating, then call INHERITED Resize with invalidation.

### UMAUtil

1. Added the file UMAUtil.a which is a combination of the old UMacApp.a and UTEView.a files. Also moved ALoadMacAppSeg from UMemory.a to UMAUtil.a so that when linking an optimized 128K ROM program you no longer get the warning that UMemory.a wasn't needed.
2. Removed the routine LongerSide.
3. DefineConfiguration now uses System 6.x as the criteria for styled TextEdit.

4. FieldToString now calls the routine whose address is stored in gFieldToStrRtn. The default for gFieldToStrRtn is StdFieldToString. You may define your own FieldToString routine by stuffing its address in gFieldToStrRgn.
5. Add PBHOpenDeny and PBHOpenRFDeny support to OpenFile.
6. Got rid of the CPU constants which are no longer needed now that we use SysEnviron.

#### UMemory

1. Modified LoadMacAppSeg so that when debugging it reports the number *and name* of the segment being loaded.
2. Added the routine LoadResidentSegments.
3. Changed some segment mappings.

#### UMenuSetup

1. Changed some segment mappings.
2. Now NullMenuProc sets the menu width to zero so we don't have to call MeedCalcMenuSize as much.
3. Added MAInsertMenu, a replacement for the Menu Manager routine InsertMenu, so that the menu's color table is added when the menu is inserted.

#### UObject

1. Removed the \$B- directive as it is no longer necessary.
2. Changed some segment mappings.

#### UPrinting

1. Fixed bug in which turning off the page breaks didn't force an invalidation.
2. Changed some segment mappings.
3. Took out handling of update events in CheckButton. (Should be reinstated at a later date.)
4. In PoseJobDialog, now call gApplication.UpdateAllWindows to force the windows to get redrawn before continuing.
5. Fixed problems with page break computation in which the fixed size page meaning was reversed in the h and v directions.
6. Removed the B\$- compiler directive.

#### UTEView

1. Added the redraw parameter to SetOneStyle.
2. Fixed bug in which auto scrolling didn't work. (The problem was that ScrollBy was called with the redraw parameter set to false.)
3. The prototype TEView is created only if the compile-time variable qTemplateViews is true.
4. Replaced calls to FieldToString, etc. with calls to WrLblField.
5. Replaced flnsetLoc and flnsetSize with flnset, a rectangle giving the inset in each direction. Changed the parameters to ITEView to reflect this. Now there are size and location parameters as for other views, and an insets parameter which indicates the top, left, bottom and right margins within the view.
6. Page breaks are now computed correctly for styled TextEdit views.
7. Added GetPrintExtent to return the view's extent minus the insets (margins).
8. Changed ContinousStyle so that you can specify the range of characters to be checked.
9. Changed CalcSelLoc to return a rectangle rather than a point.
10. The TextEdit interfaces missing in MPW 2.x are now included in UTEView.p rather than in a separate file.
11. TTEStyleCommand.Redolt now simply calls Dolt.
12. ITEStyleCommand now sets up the mode for old TextEdit as well as styled TextEdit.

13. When pasting we now avoid calling SetStylScrap unless the TEView is styled.
14. Modified StuffText so that it doesn't set fSavedTEHandle to fText if they are equal.
15. Fixed bug in SetOneStyle in which it didn't set fHTE's lineHeight and fontAscent in the non-styled case.
16. Minor changes to segmentation.
17. Fixed bugs in which the selection didn't always get scrolled into view. This was particularly noticable when typing to the bottom of the view's window.
18. CalcRealHeight now correctly computes the view's height.
19. SetOneStyle now pays attention to the mode even in the non-styled case.
20. Updated the Fields methods.
21. Fixed bugs in which the use of TESelView and TEAutoView was not conditional on the presence of 128K ROMs.
22. Modified Resize so that it forces a redraw if fAutoWrap is true and the width and height of the view changes.
23. Fixed bug in BelnScroller in which it didn't call INHERITED BelnScroller. This caused the scroller to not reset its translation values.
24. Fixed bug in GivePasteData in which it didn't ensure the view was the styled type before trying to get 'styl' scrap data. This caused erroneous errors to be reported.

### UTrace

1. Fixed bug in TRCException in which we ignored errors \$7000 thru \$7FFF instead of \$7F00 thru \$7FFF.

### UViewCoords

1. All routines are now coded in assembly language.

### MacApp Resource Files

1. Added 'res!' resources for the resident MacApp segments and resident debugging segments.
2. Added 'view' template for the Clipboard window.
3. Changed the TEView resource template to replace the old InsetLoc and InsetSize points with an Inset rectangle.
4. Got rid of the ListItem template as it was not used.

### MABuild and Make Files

1. Modified to support MPW 3.0.
2. Make is now passed the -w option to suppress warnings. This prevents warnings if Make variables are defined in the application's make file as well as MacApp's.
3. Modified the segment mappings to prevent overload of the BBRes segment and to map TEInit to GInit.
4. Added the qProceduralViews, qTemplateViews, and qWriteTemplates variables to MacApp.make1. These are passed to the Pascal compiler and govern the compilation of view l-methods, and the IRes, WRes and WriteRes methods.
5. Added UTEView.p to UDialog's dependencies.
6. Fixed bug in which not all of the object files were dependent on the setting of the ROM128K flag.
7. Added the -bf option to the link options.

### All Samples

1. Modified the 'SIZE' resources to be compatible with MPW 2.0 and MPW 3.0.

2. Modified the copyright notices in the "About" boxes to conform to Apple legal requirements.

#### Calc Sample

1. Calc now uses the GridView building block.
2. Calc now creates its views from templates.
3. Corrected path name to SaneLib.o in Calc.make.
4. Fixed various bugs.

#### DemoDialogs Sample

1. Added color to the popup menu example in the 'Views by Template' window.
2. Now use default class names in templates.
3. Set the appropriate targets for the various dialog examples. This fixed bugs in which keystrokes were not being handled properly by some of the dialogs.
4. Changed 'Nested Scrollers' to 'Side-by-Side' scrollers.
5. Added 'seg!' resource to reserve memory for the maximum code resource usage.
6. Popup menu examples are not created if the environment does not support them.
7. Increased the number of master pointers initially created to prevent heap fragmentation.

#### DemoText Sample

1. Mapped the ARes segment to Main so that the code in ARes is made resident.
2. Now use default class names in the templates.
3. Made the 'SIZE' and 'seg!' resources reasonable.
4. Shortened the prompt in the "Get Background Color" dialog, as the previous one didn't fit.
5. Fixed bug in which the font name of the selected font wasn't always checked.
6. Changed default font size to 12.
7. Fixed bug in which the low memory global TESysJust was used on 64K ROM machines, in which case it didn't exist.

#### DrawShapes Sample

1. Mapped the ARes segment to Main so that the code in ARes is made resident.
2. Corrected the 'seg!' resource.
3. Changed the 'mctb' resource id to match the corresponding 'menu' so that the Menu Manager automatically loads the 'mctb' resource.

#### Nothing Sample

1. Modified the 'SIZE' resource to be reasonable.
2. Modified TNothingView.Draw to do a PenNormal at the end.

#### PatView Sample

1. Corrected the "About" box and creator.
2. Added HandleFinderRequest override to prevent attempt to open files on Finder launch.
3. Now stagger the windows.

#### Puzzle Sample

1. Modified the MultiFinder partition size to be more reasonable.
2. Added a 'seg!' resource to identify the maximum code resource usage.
3. Added command-key equivalents to the File menu.
4. Fixed bug in "About" box.

TwoDockKinds Sample

1. Modified the MultiFinder partition size to be more reasonable.

# 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.

## 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, TCtrlMgr, 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 WRITELNS 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 changing so we cannot guarantee 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 the view'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 associated 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-explanatory. 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 `CreateAPIView` for each view in the resource. `CreateAPIView` 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 `CreateAPIView` 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 `CreateAPIView`.

## MultiFinder Support

While programs built with `MacApp 1x` 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. Referring 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

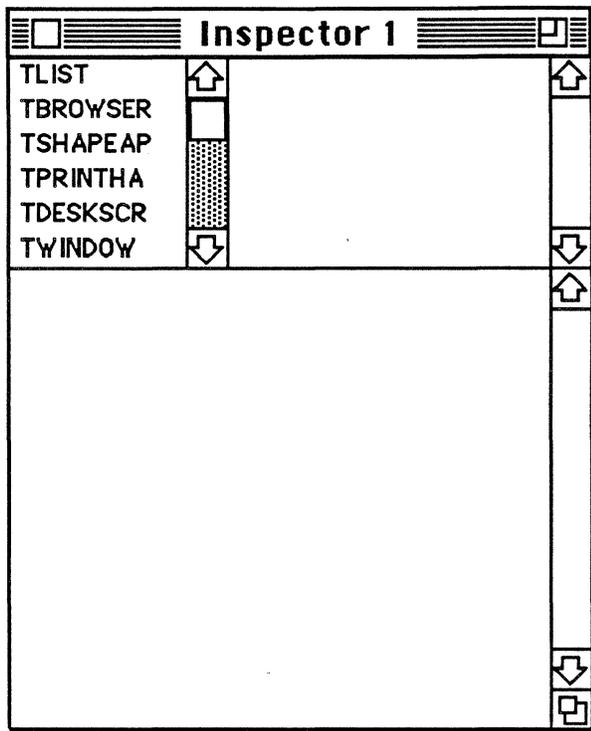
UGridView is a new building block in MacApp 2.0. It contains a set of view classes that are organized as rows and columns of cells, much like a spreadsheet. Examples of its use can be found in the samples `DemoDialogs` and `Calc`. The Inspector debugging windows also use UGridView. See the `GridView`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

## Object Inspecting

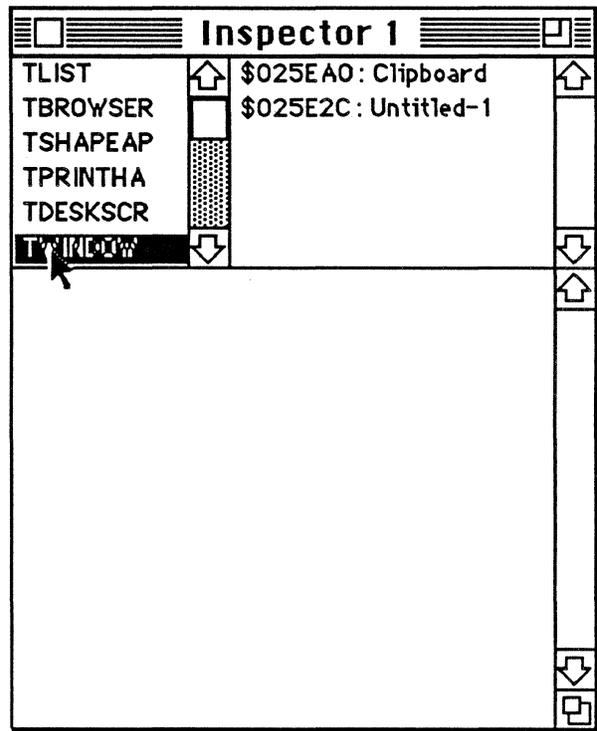
An object inspector has been added to MacApp's debugging facilities. The inspector provides an easy way to display the fields of any instantiated object. Each time the `New Inspector Window` command of the `Debug` menu is selected, a new inspector window is displayed. An inspector window is shown on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

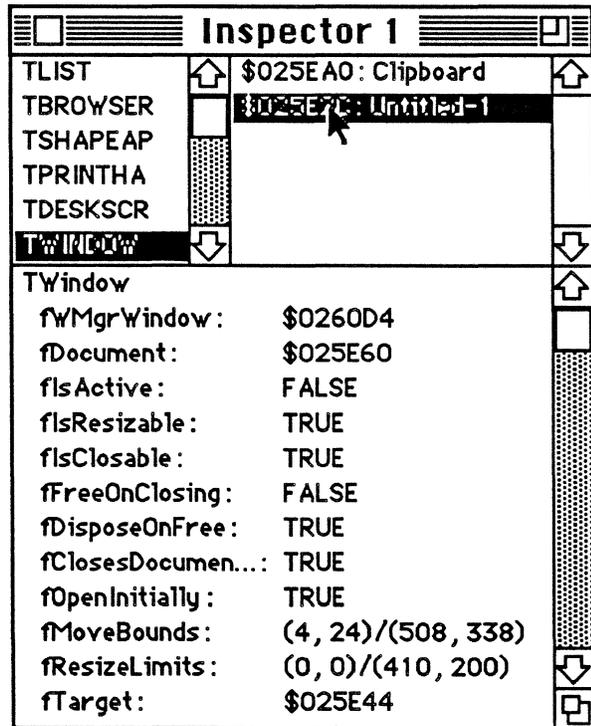
Note that the bottom pane is not automatically updated when changes occur in the data being displayed. (Refreshing the window, or clicking the object in the upper-right pane will reflect the object's current values.)



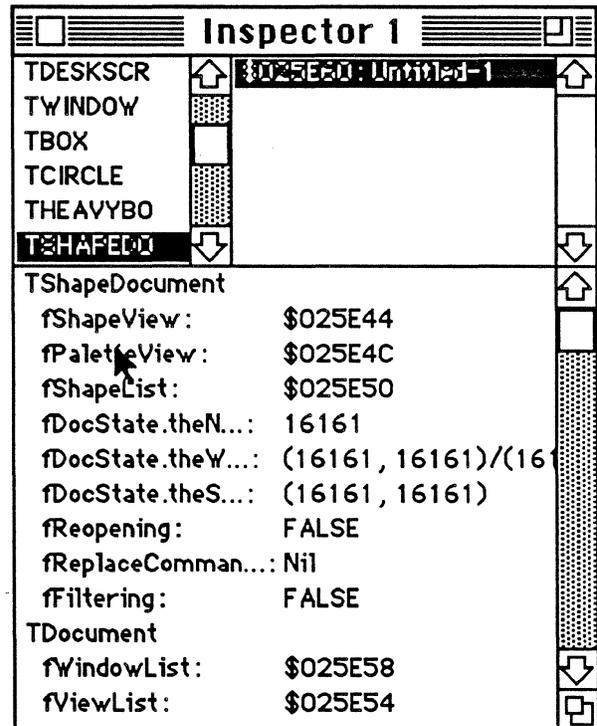
After selecting the New Inspector Window command from the Debug menu



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

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`:

<code>bInteger</code>	<code>= -1;</code>	an <code>INTEGER</code> , converted to a decimal string.
<code>bHexInteger</code>	<code>= -2;</code>	an <code>INTEGER</code> , converted to a hexadecimal string.
<code>bLongInt</code>	<code>= -3;</code>	a <code>LONGINT</code> , converted to a decimal string.
<code>bHexLongInt</code>	<code>= -4;</code>	a <code>LONGINT</code> , converted to a hexadecimal string.
<code>bString</code>	<code>= -5;</code>	a <code>STRING</code> of any length.
<code>bBoolean</code>	<code>= -6;</code>	a <code>BOOLEAN</code> .
<code>bChar</code>	<code>= -7;</code>	a <code>CHAR</code> .
<code>bPointer</code>	<code>= -8;</code>	a pointer of any type, converted to a hexadecimal string.
<code>bHandle</code>	<code>= -9;</code>	a handle of any type, converted to a hexadecimal string.
<code>bPoint</code>	<code>= -10;</code>	a <code>QuickDraw Point</code> .
<code>bRect</code>	<code>= -11;</code>	a <code>QuickDraw Rect</code> .
<code>bObject</code>	<code>= -12;</code>	a reference to another object.
<code>bByte</code>	<code>= -13;</code>	a single byte, converted to a decimal string.
<code>bCmdNumber</code>	<code>= -14;</code>	a field of type <code>CmdNumber</code> .
<code>bClass</code>	<code>= -15;</code>	announces the start of a class's fields.
<code>bOSType</code>	<code>= -16;</code>	a field of type <code>OSType</code> , converted to a 4-character string.
<code>bWindowPtr</code>	<code>= -17;</code>	a <code>WindowPtr</code> .
<code>bControlHandle</code>	<code>= -18;</code>	a <code>ControlHandle</code> .
<code>bTEHandle</code>	<code>= -19;</code>	a <code>TEHandle</code> .
<code>bLowByte</code>	<code>= -20;</code>	the low byte of a 2-byte word (integer), converted to a hex string.
<code>bHighByte</code>	<code>= -21;</code>	the high byte of a 2-byte word (integer), converted to a hex string.
<code>bPattern</code>	<code>= -22;</code>	a <code>QuickDraw Pattern</code> .
<code>bFixed</code>	<code>= -23;</code>	a field of type <code>Fixed</code> .
<code>bRgnHandle</code>	<code>= -24;</code>	a <code>RgnHandle</code> .
<code>bRGBColor</code>	<code>= -25;</code>	a field of type <code>RGBColor</code> .
<code>bTitle</code>	<code>= -26;</code>	field data is ignored—only its title is displayed in an Inspector window.
<code>bGrafPtr</code>	<code>= -27;</code>	a pointer to a <code>QuickDraw graf port</code> .
<code>bStyle</code>	<code>= -28;</code>	a <code>QuickDraw Style record</code> .
<code>bVCoordinate</code>	<code>= -29;</code>	a <code>VCoordinate</code> .
<code>bVPoint</code>	<code>= -30;</code>	a <code>VPoint</code> .
<code>bVRect</code>	<code>= -31;</code>	a <code>VRect</code> .
<code>bFontName</code>	<code>= -32;</code>	a font number, converted to a font name.
<code>bStringHandle</code>	<code>= -33;</code>	a handle to a string.
<code>bCntlAdornment</code>	<code>= -34;</code>	a field of type <code>CntlAdornment</code> .
<code>bIDType</code>	<code>= -36;</code>	a field of type <code>IDType</code> , converted to a 4-character string.
<code>bResType</code>	<code>= -37;</code>	a field of type <code>ResType</code> , 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 addresses 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 'DEBUG' and id 300. The format of the 'DEBUG' resource is:

```
type 'debug' {
    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

<code>TView</code>	An <i>abstract class</i> (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.
<code>TWindow</code>	A subclass of <code>TView</code> that implements a Toolbox window. The window may be modal or modeless.
<code>TScroller</code>	A subclass of <code>TView</code> that is able to "scroll" its contents by effecting a coordinate transformation.
<code>TControl</code>	A subclass of <code>TView</code> , <code>TControl</code> 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.
<code>TCtlMgr</code>	A subclass of <code>TControl</code> , <code>TCtlMgr</code> is an abstract class that implements Control Manager controls.
<code>TScrollBar</code>	A subclass of <code>TCtlMgr</code> that implements scroll bars generically.
<code>TSScrollBar</code>	A subclass of <code>TScrollBar</code> that implements scroll bars specifically for scrolling MacApp views.

### UTEView View Classes

<code>TTEView</code>	A subclass of <code>TView</code> that implements a text edit view based on the ROM TextEdit.
----------------------	----------------------------------------------------------------------------------------------

## UDialog View Classes

TButton	A subclass of TCtrlMgr that implements push buttons.
TRadio	A subclass of TCtrlMgr that implements radio buttons.
TCheckBox	A subclass of TCtrlMgr 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 <code>TEView</code> 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.

