MacApp® 2.0b12

Release Notes
March 5, 1990

QuickStart

This release of MacApp requires MPW 3.1 or 3.0 final. It will not work with MPW 2.0.2 and beta versions of MPW 3.0. *If your computer has enough memory, setting the RAM Cache to 64K will speed up MPW significantly*.

Important Notes to b5 Users

These release notes describe the changes since 2.0b5. It includes sections summarizing the changes in particular areas of MacApp, followed by a detailed changed list. The summarized sections describe *important* changes to MacApp. Make sure you read them as they tell you what to change in order to move your b5 application to this release.

Installing

To install MacApp on your system, do the following:

1. On your hard disk, create a folder named MacApp.

2. Copy all files/folders from all MacApp disks into the MacApp folder.
NOTE: If you only intend to use the pre-built MacApp® libraries
or do not intend to build the Examples, then only copy the first 3 disks.

3. Move the contents of any folder that is titled "X!" to folder "X".

4. Drop a copy of UserStartup•MacApp into your MPW folder.

5. Launch MPW. UserStartup•MacApp will be automatically executed. If the MacApp folder cannot be automatically located, you will be asked to find it.

6. Optionally, create a folder to hold your Application source files, Rez input file, and make file and place it in the MacApp folder. The naming conventions for your application's files are:

M*YourApp*.p {The optional Main program}
U*YourApp*.p {The optional interface part of your application unit}
U*YourApp*. .p {The optional include file[s] that contain the implementation of the unit. The
portion of the name[s] can be anything though it is often inc1. }

OR --------------------
YourApp.p {The Main program. See Nothing.p for an example. }

AND -------------------
YourApp.r {The optional Rez input file. Default.r will be used if you do not supply an input

file }
YourApp.MAMake {The optional Make file. }

You can copy all of these files at once in MPW by typing:

```
duplicate "Source Volume: YourApp " "Dest Volume:"
```

If you prefer to use a different naming convention, you may have to modify the MacApp build system.

Building

Build Instructions:

1. You may need to create a MAMake file. See the MacApp Reference Manual, or the Calc sample program to learn how.

2. In MPW, set your directory to your Application folder:
   directory "{MacApp}MyApp Folder"

3. In MPW, type
   MABuild YourApp

   where YourApp is the name of your application. The neccessary files (all of them, the first time) will be compiled, the MacApp libraries are already prebuilt for normal debug and non-debug variations, and your application will be linked. Progress information will be output to the screen as the build proceeds.

The first time you build an application, Rez will run after the link to create your application's resources. Rez will not run again unless you change the Rez input file (YourApp.r).

The Examples folder contains six folders containing sample MacApp programs. Each of the sample programs can be built by executing the following commands:

```
MABuild "{MAExamples}Calc:Calc"
MABuild "{MAExamples}Cards:Cards"
MABuild "{MAExamples}DemoDialogs:DemoDialogs"
MABuild "{MAExamples}DemoText:DemoText"
MABuild "{MAExamples}DrawShapes:DrawShapes"
MABuild "{MAExamples}Nothing:Nothing"
```

The following command also builds all of the sample programs in the Examples folder:

```
#Select and execute the following lines to build all the examples at once and
# skip building of MacApp
		MABuild
		"{MAExamples}Calc:Calc"
		"{MAExamples}Cards:Cards"
		"{MAExamples}DemoDialogs:DemoDialogs"
		"{MAExamples}DemoText:DemoText"
		"{MAExamples}DrawShapes:DrawShapes"
		"{MAExamples}Nothing:Nothing"
		-nofail -debug
```

All debug flags default to FALSE so you will not have a debug window in your application unless you specify -Debug. Any WriteLn's will appear in this window. You can also do ReadLn's from this window.
(bugs: You cannot write a packed array of char. You cannot do a readLn without the Ln, i.e. Read(ch) doesn't work)

Note that this is changed from previous releases of MacApp where -Debug was the default. If you wish to change your default follow the instructions found in the General Reference. MacApp Debugger instructions can be found in the MacApp reference manual.

'cmnu' Resource Type

The file MacAppTypes.r defines a special type of menu template called a 'cmnu' (command menu). It has the same definition as a MENU with the addition of a command number for each menu item. After Rez has run, a program called PostRez runs that converts these into MENUs and extracts the command numbers for use by MacApp.

Help

The contents of the file: "Insert to MPW.Help" can be pasted into your MPW.Help file in order to give automatic access to the MacApp.Help file in the same way as the CPlus entry in MPW.Help can give access to the CPlus.Help file. To use it just select the contents of: "Insert to MPW.Help" and paste into the alphabetically correct place in the MPW.Help file. Future releases of MPW will include this automatically.

Also, the MABuild tool has a Commando and a -Help option.

__Changes__

MacApp Folder Organization and Shell Environment Variables

The organization of the MacApp folder has changed. The biggest difference is that the interface files have been separated from implementation files. Following MPW's lead we now have an "Interfaces:" folder, which contains folders for assembly includes ("AIncludes:"), Rez includes ("RIncludes:"), and Pascal interfaces ("PInterfaces:"). The following chart describes the MacApp folder hierarchy, and the shell variables containing the folder's path name:

MacApp:	"{MacApp}"
Tools:	"{MATools}"
Interfaces:	"{MAInterfaces}"
AIncludes:	"{MAAIncludes}"
RIncludes:	"{MARIncludes}"
PInterfaces:	"{MAPInterfaces}"
Libraries:	"{MALibraries}"
Examples:	"{MAExamples}"
Calc:	
Cards:	
DemoDialogs:	
DemoText:	
DrawShapes:	
Nothing:	

Object code for the MacApp libraries is created in a sub–folder of the "{MALibraries}" folder. Similarly, object code for your application will be put in sub–folders of your application's folder.

The 2.0b5 shell variables representing pathnames in MacApp are no longer defined, in favor of a new set of shell variables. Here is a description of how the b5 folder organization maps onto the new organization:

MacApp:MacApp Source Files: "{SrcMacApp}"

> Unit interfaces now found in "{MAPInterfaces}", implementation code now found in "{MALibraries}".

MacApp:MacApp Object Files: "{ObjMacApp}"

> Object code is now placed in folders in "{MALibraries}".

MacApp:MacApp Resource Files: "{RezMacApp}"

> The files MacAppTypes.r and ViewTypes.r are now located in "{MARIncludes}". The rest of the Rez files are located in "{MALibraries}". The compiled (.rsrc) resource files are located in the object file folders.

MacApp:MacApp Make Files: "{MakeMacApp}"

> The MacApp make files are located in "{MATools}". The files MacApp.make1, MacApp.make2, MacApp.opt.make2, and MacApp.make3 have been replaced with "Basic Definitions" and "Build Rules and Dependencies".

MacApp:MacApp LOAD Files:	"{LoadMacApp}"
	Load files are not used by MPW 3.0 so this folder is no longer needed.

Obviously you need to change references to 2.0b5 shell variables (such as "{SrcMacApp}") that are no longer defined. A more comprehensive list of shell variables defined by MacApp is included in the MacApp Interim Manual.

MABuild

MABuild has been completely rewritten (in Object Pascal), with many new features. Among these features is the ability to maintain separate object code for any number of build options you desire (including debug and non-debug), and the ability to build applications requiring specific system or hardware features such as system 6.0, a 68020 or an FPU (floating-point unit such as the 68881). The MacApp 2.0 General Reference describes these changes in some detail. Unlike 2.0b5, MABuild will not build MacApp itself unless asked. MABuild runs faster if it doesn't have to check that MacApp is built.

You can obtain a list of MABuild options by typing "MABuild -help". A Commando interface is also provided for MABuild.

To get started, here are the basic MABuild options:

MABuild *appname* -debug	builds a debug version of your application, putting the object code and executable application in the ".Debug Files:" folder in your application's folder.
MABuild *appname* -nodebug	builds a non–debug version of your application, putting the object code and executable application in the ".Non-Debug Files:" folder in your application's folder.
MABuild *appname* -debug -autobuild	builds a debug version of MacApp and your application, putting the MacApp object code in the ".Debug Files" folder in "{MALibraries}", and putting your object code and executable application in the ".Debug Files:" folder in your application's folder. You can omit *appname* if you wish to build MacApp only.
MABuild *appname* -nodebug -autobuild	builds a non–debug version of MacApp and your application, putting the MacApp object code in the ".Non-Debug Files" folder in "{MALibraries}", and putting your object code and executable application in the ".Non-Debug Files:" folder in your application's folder. You can omit *appname* if you wish to build MacApp only.

MABuild has been rewritten again and is now about 99% in Object Pascal and much faster. MABuild has some new compiler options. By adding "-CPlusSupport" to the command line, you tell MABuild that you are compiling C++ units, even if they don't conform to the normal MABuild naming convention. By adding "-PasLoad" and/or "-CPlusLoad," you can have MABuild put precompiled headers into a separate folder. These headers will be separated according to your set of compiler options, just like your object code, so the appropriate set will be available when you need them.

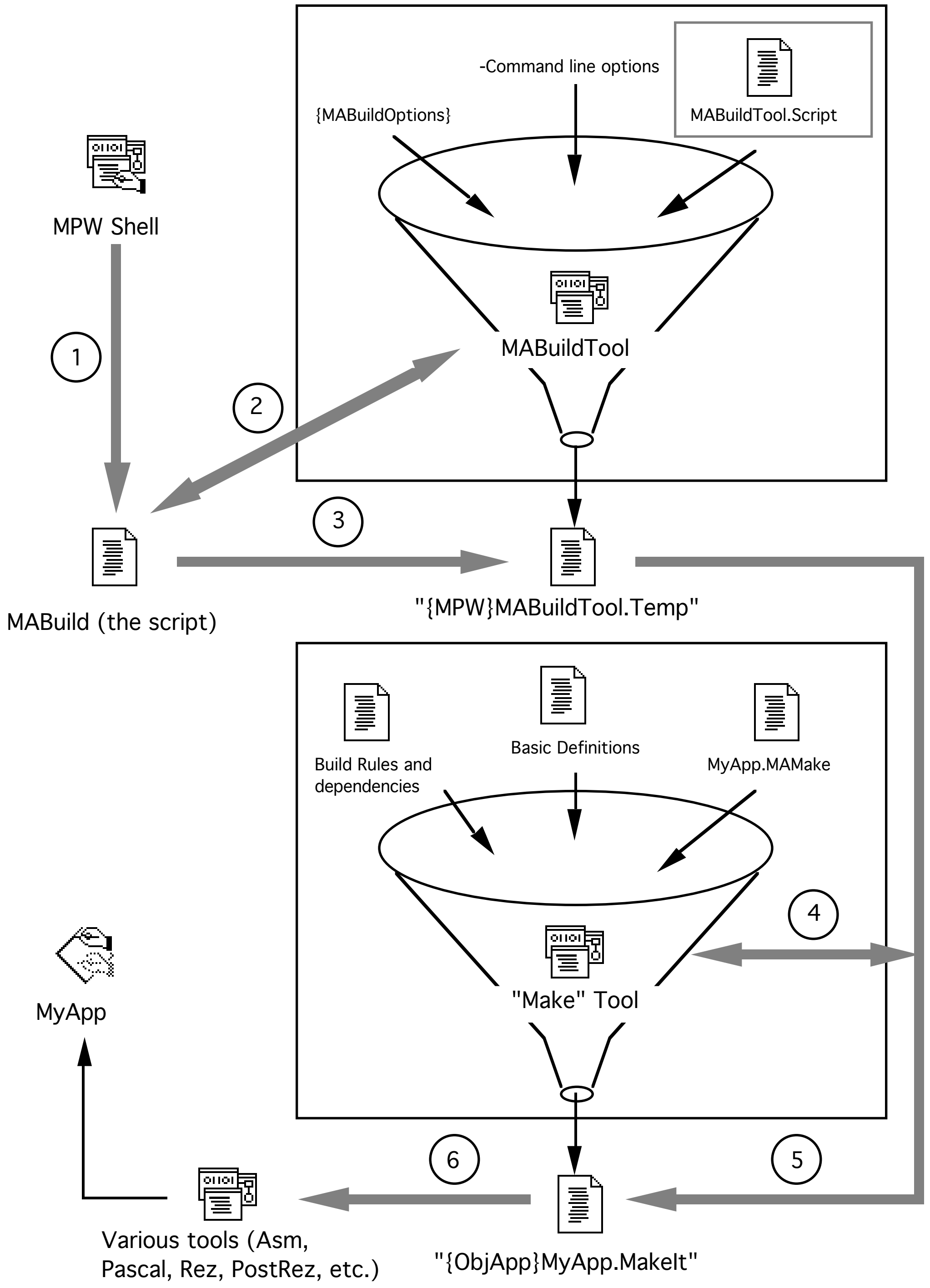


Figure 1 - MABuild flow of control

Make Files

You must rename your make files to have a '.MAMake' suffix instead of '.make'. *This change* *allows the MPW Build menu commands to build MacApp programs.*

The shell variables used in your MAMake file are different than before. Here's how to convert most cases:

- Uses of "{SrcMacApp}" should be changed to "{MAPInterfaces}".
- Uses of "{ObjMacApp}" should be changed to "{MAObj}".
- References to any of your own source files should be preceded with "{SrcApp}".
- References to any of your own object code or .rsrc files should be preceded with "{ObjApp}".

In MacApp 2.0b5 there was a standard set of Make variables that each application's make file defined. Now these variables are defaulted and it is only necessary to define them in your MAMake file if you wish to override the default. Here is a list of the variables and how they may have changed since b5:

AppName	Stays the same.
Creator	This is the same as before. (Actually it doesn't currently do anything. The only way to set the application's creator is with a bundle resource, as is done in the Calc sample. In the future we hope to use this variable to set the creator.)
NeededSysLibs	You should probably remove this from your MAMake file. If omitted then the following set of system libraries is linked with your application: NeededSysLibs = {PascalSupport} {CPlusSupport} {PerformLib} "{Libraries}Interface.o" "{Libraries}ToolLibs.o" {NonFPUSANELib} {FPUSANELib} Any of these libraries not required by your application will produce a warning by the Linker, but otherwise be unused. Rather than redefine NeededSysLibs in your MAMake file, we suggest that you list additional link files in the OtherLinkFiles variable.
BuildingBlockIntf	This can be omitted. It is a list of the building block interfaces your application depends on. If any building block in the list changes, then your application will be rebuilt. If you leave it in, make sure you change the "{SrcMacApp}" references to "{MAPInterfaces}" as was done in Calc's MAMake.
BuildingBlockObjs	This is no longer necessary and should be taken out of your MAMake file. (Unlike MacApp 2.0b5, all building blocks are combined with MacApp's libaries to form one big library with which your application is always linked.)
OtherInterfaces	The same as before, but make sure you use "{SrcApp}" for files in your application's folder. (See Calc.MAMake for an example.)
OtherLinkFiles	The same as before, but make sure you use "{ObjApp}" for object files whose sources are in your application folder. (See

	Calc.MAMake for an example.)
OtherSegMappings	The same as 2.0b5.
OtherRezFiles	The same as before, but make sure you use “{SrcApp}” for files in your application’s folder. (See DemoDialogs.MAMake for an example.)
OtherRsrcFiles	Omitting this from your MAMake file makes your application dependent on all of MacApp’s .rsrc files. This is not the same as using them all–-you decide which ones to use by including them in your application’s .r file. Generally this can be omitted unless your application is dependent on other .rsrc files. (See Calc.MAMake for an example.)

Uses Statements

The MacApp libraries now use an include file mechanism similar to that used by the Pascal system interface files. Besides speeding up the build process it enables you to simplify your USES statements. Here is the new USES statement:

```
USES
  { • MacApp }
  UMacApp,

  { • Building Blocks }
  UPrinting, UTEView, UDialog, UGridView,

 { • Implementation Use }
 list any units or system interfaces your unit requires in its implementation
```

Using UMacApp automatically includes the interfaces for all units required by UMacApp’s interface, including UObject, UList, etc. It replaces the long list of Toolbox and MacApp unit interfaces required in 2.0b5. The building block declarations are of course optional. You should list UMacApp before any of the building blocks or before any other units for optimal compile performance.

Unlike MacApp 2.0b5, MacApp 2.0b9 does not include the interface to the entire Toolbox. When you compile your program you may find that some of the Toolbox identifiers you use are no longer defined. To remedy this, add the required unit name to your USES statement (e.g. add Resources if you use a Resource Manager routine). If you are unsure which unit should be added you can search the “{PInterfaces}” files for the needed identifier, thereby locating the unit to be used.

(You may notice that MacApp’s interfaces now use the “include” mechanism used by the MPW Pascal interfaces. And you may be tempted to use this technique for your own units. **Don’t**. The compiler is incapable of handling more than a limited number of nested includes. If you use this technique you will exceed the limit very quickly. A future release of the compiler will increase this limit.)

Program Initialization

The initialization sequence of a MacApp program has changed somewhat. To retain the same behavior as you had in 2.0b5, simply replace the call to InitToolBox in your main program with a call to InitUMacApp. However, you may wish to take advantage of some new capabilities.

It is now possible to initialize the Toolbox and MacApp separately. The advantage in doing so is that once the Toolbox is initialized you can display a “splash screen” or verify that the system is capable of running your application before continuing.

Here then is the way to initialize and run a MacApp 2.0 application (or if you would rather read code, look at the file MCalc.p):

1. Make sure your main program can run on any hardware/software configuration. You do this by including the {$MC68020-} and {$MC68881-} compiler directives before your Uses statements in your main program. Your main program should not include any code of its own other than that necessary to initialize and start your application.

2. Call InitToolBox. This performs the required Macintosh Toolbox initialization as described in Inside Macintosh, and it calls DefineConfiguration to determine the characteristics of the machine your application is running on. All of this is guaranteed to work on any Macintosh system.

3. Call ValidateConfiguration and/or do your own system validation. ValidateConfiguration returns true if the system configuration is capable of running your application according to the MABuild options in effect when the application was built. For example, if you specified -NeedsMC68020 when building your application then ValidateConfiguration returns false if the application is run on a Macintosh Plus. ValidateConfiguration will run on any system. Your application may need to do additional checking if it has unusual requirements. You can look at the source of ValidateConfiguration (in UMacAppUtilities) to see what it checks.

4. If ValidateConfiguration returns false, or your tests (if any) are negative, then display an appropriate alert with StdAlert and terminate the program. MacApp supplies a generic alert for this purpose, whose id is defined by the constant phUnsupportedConfiguration. You can use that, or even better would be to define your own alert that is specifically for your application.

   If ValidateConfiguration returns true and your tests are positive, then continue with your application.

5. If you want to display a “splash screen” while your program is starting up, now’s the time to do it. First call PullApplicationToFront to ensure that the splash screen window is shown in front of all other application windows, then show your splash screen. Since MacApp has not been initialized yet, you’re restricted to using the facilities of the Toolbox at this point. A Dialog Manager window is probably your best bet. The Calc sample program shows an example of this.

6. Call InitUMacApp to initialize MacApp.

7. Initialize any building blocks you use by calling InitUGridView, InitUDialog, InitUTEView, and /or InitUPrinting.

8. Create and initialize your application object.

9. If you displayed a splash screen, get rid of it.

10. Call your application object’s Run method.

gFrontWindow and gDocument

These two global variables have been eliminated. To obtain the equivalent to gFrontWindow, call the application method GetActiveWindow. (There is also a GetFrontWindow method–it is *not* the same as GetActiveWindow. It is possible that a window may be the front window but not be the active window because the activate event for the window has yet to be processed.) The equivalent to gDocument can be obtained from the fDocument field of the window returned by GetActiveWindow.

UList

If you used the GetItemNumber method, read on…

GetItemNumber has been replaced by a pair of methods, GetSameItemNo and GetEqualItemNo. GetSameItemNo returns the index of a given object in a list, or *zero* if the object isn't in the list. GetEqualItemNo returns the index of any object considered to be "equal" to the given object, or *zero* if an "equal" object isn't found. In TList these methods behave the same way. That is, TList has no way to compare two object references other than to determine if they refer to the same object. In TSortedList the behavior of these methods differs: GetSameItemNo still returns the index of a given object in the list, whereas GetEqualItemNo returns the index of *any* object in the list which, when compared to the given object, causes the Compare method to return zero. Thus there may be different objects in the list for which GetEqualItemNo will return a non-zero value.

Note that if the list contains multiple references to the same object then the index returned by GetSameItemNo depends on the implementation.

The TList class has been entirely rewritten for MacApp 2.0b12. This class originally had some fundamental design flaws that caused some improper behavior. For instance, TLists didn't respond very well to having items deleted while being iterated over with TList.Each, and didn't work at all if you tried to insert new items under the same circumstances. The new class has been redesigned with this and other issues in mind.

There is now a base class called TDynamicArray. This class contains the basic methods for adding, deleting, replacing, and retrieving elements (InsertElementsBefore, DeleteElementsAt, ReplaceElementsAt, and GetElementsAt). All of these methods are designed with multiple, n-sized elements in mind. Unfortunately, due to time constraints, only full support for elements whose size are a power of two are supported. This is sufficient for supporting TLists, as each element in a TList is 4 bytes long.

ComputeAddress and GetSize were promoted from TList to TDynamicArray, and SetArraySize was added so that an array could be correctly sized in one single step, reducing the performance hit provided by the Macintosh Memory Manager when resizing the object one element at a time. InsertElementsBefore was also modified to optimize Memory Manager performance by requesting new memory in chunks. This approach, graciously provided by Peter Gaston and used with his permission, greatly speeds up the insertion of new objects.

EachElementDoTil is the TDynamicArray method which acts as the bottleneck for all list iterating. It can iterate both forwards and backwards, responding more appropriately to insertions and deletions. It does this by keeping a doubly-linked list of nodes that keep track of where it is in the iteration process. Each time EachElementDoTil is called, one of these nodes is created and inserted into the list. These nodes keep track of the index number of the item we are currently examining, the direction in which we are iterating, and the upper and lower bounds of the iteration. If something happens that could affect the index number of the element we are looking at, then TDynamicArray goes through this linked list of nodes, tweaking the index numbers appropriately. When EachElementDoTil is done, the node is removed from the list. In this way, EachElementDoTil can be re-entered as often as you like.

As with the original TList, elements of dynamic arrays are added to the end of the TDynamicArray object handle. However, access to those elements is now more encapsulated by the consistent use of the ComputeAddress method, and by adding a new method to TObject named DynamicFields. TObject.DynamicFields performs the same function as TObject.Fields, but is responsible for reporting information about the dynamic elements associated with an object. By overriding this method, one can define objects whose dynamic elements are not appended to an object, but are, for example, stored somewhere else in memory, or even on disk.

Finally, a couple of other methods were implemented in TDynamicArray. Merge can be used for inserting equivalent elements from another TDynamicArray. IsEmpty is used to determine if there are any elements in the array.

With all of this groundwork out of the way, TList has a good base upon which to build. All of the old TList methods are still in place and work the same as before (except RemoveDeletions), but now may simply be aliases to newer and more flexible methods.

For instance, FirstThat, LastThat, and Each used to be very similar to each other, implementing their own distinct loops, but starting at different locations, and iterating in different directions. Now, they all call a new method called TList.IterateTil. IterateTil takes two parameters. The first is a FUNCTION parameter that returns TRUE when the iteration process should stop. The second is a Boolean that determines which direction in which to iterate (a feature now possible through the good graces of TDynamicArray.EachElementDoTil). When IterateTil is done, it returns the TObject that caused the iteration to halt, as well as the index number of that object.

Some other methods have been added to TList to round out sets of methods that already exist. For instance, Insert has been added to complement InsertBefore, InsertFirst, and InsertLast. It’s basically a synonym for InsertLast, but is provided as a logical base method for TSortedList.Insert. Also, AtDelete has been added to round out At and AtPut.

Two other methods have been added to give stack functionality to TList. These methods are Pop and Push. They are basically synonyms for InsertLast and AtDelete, but are provided to make your code more readable.

A SortBy method has been added to sort the elements of your TList. This method takes a comparison routine, and performs a nice Shell sort. Currently, this method is only used by TSortedList.Sort, but I expect that future versions of MacApp will have the many shell sorts scattered throughout its code replaced with a call to this method.

Finally, DynamicFields is overridden to provide the MacApp Inspector with the information it needs.

View Creation From Resource Descriptions

We now have the ability to instantiate objects of any class when given a class name. This capability did not exist in previous releases of MacApp or MPW, which is why we used view “registration.” View registration is no longer required, nor is it necessary to create “prototype” objects. This simplifies the creation of views from resources and is more space and time efficient than registration.

There is one tricky issue that must be dealt with. The Linker strips code for classes that are not directly instantiated with a New call, assuming the class is not used by the application. (Note that previous versions of the Linker were unable to do this and left in the code for all classes whether you used them or not.) This feature is generally to your advantage as it allows you to link with large libraries of classes (like MacApp), leaving it to the Linker to figure out which classes are actually used.

However, if a class is instantiated only by class name then it is necessary to ensure that the Linker doesn't strip that class. The simplest way to prevent the Linker from stripping code for classes whose instances are created by class name is to include a Member call for that class, in a routine that is actually used by your application.

The following code fragments show how to convert your old view registrations.

<u>The b5 technique:</u>

```
VAR
  aMyView:                TMyView;

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

<u>The b7 - 8.1 technique:</u>

```
VAR
  aMyView:                TMyView;

 {Prevent linker from stripping code to create a TMyView}
  IF gCreateWithTemplates THEN
      NEW (aMyView);
```

<u>The new technique:</u>

```
 {Prevent linker from stripping code to create a TMyView}
  IF gDeadStripSuppression THEN
      IF MEMBER(TObject(NIL), TMyView) THEN ;
```

Provided this code is *in a routine that gets executed* (say the application object's initialization method) then the Linker will not strip the code for TMyView.

Warning: **If you fail to include a Member on all of the necessary classes, your application may work in the debug case, but fail in the non-debug case. If this happens to you check your Member statements to ensure you've included all that you need.**

Focus Assertions

In this release of MacApp we have restored the MacApp 1.1 notion of focus assertions. This helps to eliminate focusing errors (e.g. when drawing or invalidation occurs in the wrong place in a window). There is now a debug–only view method called AssumeFocused which can be called to verify that the currently focused view is SELF. Calls to AssumeFocused have been liberally sprinkled throughout MacApp, particularly in view methods that don't call Focus but expect the focused view to be set to SELF.

When you convert your b5 application you will probably experience program breaks in the debugger that start with the phrase "Failed AssumeFocused." This will be irritating because your program probably worked just fine before. But trust us. These program breaks are for your protection. I repeat: *These program breaks are for your protection*. It means that you have attempted to call a method that requires its view to be focused, and the view isn't focused. Typical, seemingly innocuous cases include calling GetQDExtent, ViewToQDPt, ViewToQDRect, QDToViewPt, and QDToViewRect. *These all rely on the focus being properly set.*

View Resources

There is one addition to the 'view' resource. That is the "include–views–at" option which allows you to include a 'view' resource at a specific location. It is similar to the include option with the addition of a point, which is used to offset all included views whose parent id is "root." See DemoDialogs.r for an example.

Resource Files

If you were using MPW 2.0.2 or 3.0 beta, the format of the 'size' resource has changed yet again. You'll have to change yours accordingly (check the MacApp samples).

You need to remove the MacAppRFiles variable from your include statements, as shown below:

The b5 includes:

```
#ifdef Debugging
include MacAppRFiles"Debug.rsrc";
#endif
include MacAppRFiles"MacApp.rsrc";
include MacAppRFiles"Printing.rsrc";            /* If you use UPrinting */
include MacAppRFiles"Dialog.rsrc";              /* If you use UDialog */
```

The new includes:

```
#if qDebug
include "Debug.rsrc";
#endif
include "MacApp.rsrc";
include "Printing.rsrc";    /* If you use UPrinting */
include "Dialog.rsrc";      /* If you use UDialog */
```

Please be aware that resource ids up through 999 are reserved for use by MacApp. Your resource ids should be 1000 or higher.

The names of the meta-variables passed to Rez are now consistent with those passed to the Pascal compiler. For example, if the build is for debugging then the Rez meta-variable qDebug is set to true, just as it is passed to the Pascal compiler. (For now, the 2.0b5 technique of looking for the *definition* of the meta–variable Debugging is still supported.)

Resource files are now required to include their own Types files to get resource type definitions. This saves time on multiple rezzes where many of the files don't require the full suite of Types files to define the few types they use. **NOTE: your resource files will not compile until you make this change.**

UDialog

In MacApp 2.0b5 it was necessary to call the dialog view’s SelectEditText (or DoSelectEditText) method in order to indicate which edit text item should be selected when the dialog is opened. Now this is done automatically provided the window resource’s fTargetId contains the id of the edit text view to be selected. So if you set this in your ‘view’ resources you should be able to eliminate the SelectEditText call.

The handling of text field validation has changed. The Validate method (used by TNumberText for example) now returns a longint result which indicates the reason the text is invalid, or returns noErr (zero) if the text is valid. Previously Validate simply returned a boolean value indicating whether the text was valid. TEditText.StopEdit is now a procedure that terminates the editing of a field without attempting to validate it.

TDialogView.CantDeselect now displays an alert when the text is invalid. The text of the alert is based on the result of the Validate method. After the alert is displayed the edit text is set to its previous value. The hope is that you can change the response to invalid text simply by overriding CantDeselect (e.g. you may wish to change the alert or choose not to reset the text) without overriding TEditText and its descendants.

Other changes: TEditText.StopEdit is now a procedure that terminates the editing of a field without attempting to validate it. TDialogView.DeselectCurrentEditText has been added–it may be overridden to eliminate validation of text fields when tabbing.

UGridView

This unit has undergone some fairly substantial changes. The result is improved performance, reduced size, and more functionality. Note that it is now necessary to call the routine InitUGridView during the initialization of your application.

There were some architectural changes to the classes in this unit, which will have little effect on most programs. A new class has been defined, TRunArray, which encapsulates the maintenance of the run arrays used for column widths and row heights. In TGridView, the methods DoHilite and CreateHighlightRgn roughly translate to the new methods HighlightCells and CellsToPixels. Note that HighlightCells is passed a region of cells rather than a region of pixels, and it is *always* called when cells are highlighted, even when highlighting cells as part of mouse–tracking. This enables you to override one method to change cell highlighting, and to perform custom highlighting on individual cells much more easily than before. Furthermore, the conversion of regions of cells to pixels has been greatly sped up for non-rectangular cases.

The cell selector command objects have been completely rewritten and simplified. TRowSelectCommand and TColumnSelectCommand are now implemented.

UMemory

A major bug was discovered in this unit, one which has been there ever since MacApp 2.0ßx. It turns out that in the non-debug, 128K ROM case the LoadSeg trap patch was never installed. Because of this, segments were never unloaded, and MacApp's much-vaunted memory management scheme was seriously crippled. Now that segments do unload, this could expose areas of your code which incorrectly pass relocatable items as parameters. This includes passing fields of unlocked objects as VAR parameters, and passing pointers to relocatable memory, such as fields greater than 4 bytes.

UMacApp

MacApp’s event handling code has been greatly enhanced through the addition of a TCommand queue. This command queue augments the existing method of fetching events from the system with GetNextEvent or WaitNextEvent. Applications add to this queue with PostCommand, and MacApp retrieves commands from the queue with GetNextCommand (these two methods are actually TEvtHandler methods, but are overridden by TApplication).

The command queue is supported by MacApp in two places. TApplication.HandleEvent checks this queue after your TEvtHandlers have been given a chance to handle an event. If they don't return a TCommand back to HandleEvent (in other words, they return gNoChanges or NIL (BTW you can return NIL instead of gNoChanges now if you wish)), the command queue is checked. If a TCommand is there, it is popped off and executed as if it had been returned by your TEvtHandlers. This is handy for cases where you would like to implement some action through a TCommand, but are unable to return a TCommand from the code you are currently executing. For instance, **many** people have mentioned in passing that they would like to be able to return a TCommand from TView.DoChoice. Some people have even gone so far as to modify their versions of MacApp so that they can do so. With the TCommand queue, all you have to do is create your command, and post it with PostCommand. MacApp will pop that command off and execute it the next chance it gets.

The command queue is also checked in TApplication.PollEvent. Before TApplication.GetEvent is called, the command queue is checked. If there are any commands ready to go, then they are pulled off and sent to TApplication.PerformCommand. Otherwise, GetEvent is called as before.

The command queue is more than just a simple array of TCommands. There is a three level process that GetNextCommand goes through in determining which TCommand to return to you.

First, it checks to see if the command is ready to be executed. When posting a command, it is possible to put it into a dormant state, as determined by its IsReadyToExecute method. If IsReadyToExecute is FALSE, GetNextCommand looks for another TCommand to return to you. If IsReadyToExecute is TRUE, GetNextCommand returns that TCommand to you. This allows you to initialize and install commands into the queue, and have them executed at a later time simply by setting fReadyToExecute to TRUE (or by overriding IsReadyToExecute to check some external condition).

Secondly, the TCommands in the queue can be prioritized. There are 127 priorities you can assign, and five of them have been symbolically defined for you: kPriorityLowest, kPriorityLow, kPriorityNormal, kPriorityHigh, kPriorityHighest. These five should be sufficient for most applications. When you post a TCommand with PostCommand, it is inserted into the queue ahead of all other TCommands with lower priorities.

Finally, GetNextCommand will leave a command in the queue if that command's fRecurring field is TRUE. This allows you to keep a command around if you expect to need to use it again at a later time, or would like to have it executed again immediately.

So what is all of this good for? One example was already shown. It is sometime desirable to return a command from TDialogView.DoChoice. But TDialogView.DoChoice doesn't let you pass a TCommand back. By using gApplication.PostEvent, you can create a command and have it executed the next time through the main event loop.

It would also be possible for you to trigger a command from an interrupt task. Say that you had some sort of VBL or Time Manager task that was on the lookout for some set of conditions. When these conditions are met, that task could set the fReadyToExecute field of an already queued command. The next time through the event loop, this command would be executed (the command would have to be locked down though).

Finally, the command queue also lays down the groundwork necessary for document-level undo. Since GetNextCommand and PostCommand are TEvtHandler methods, TDocument could override them to implement a command queue specific for that document. This is something that is being looked at for future versions of MacApp.

UInspector

The appearance of the Inspector windows has been changed slightly (Figure 2).

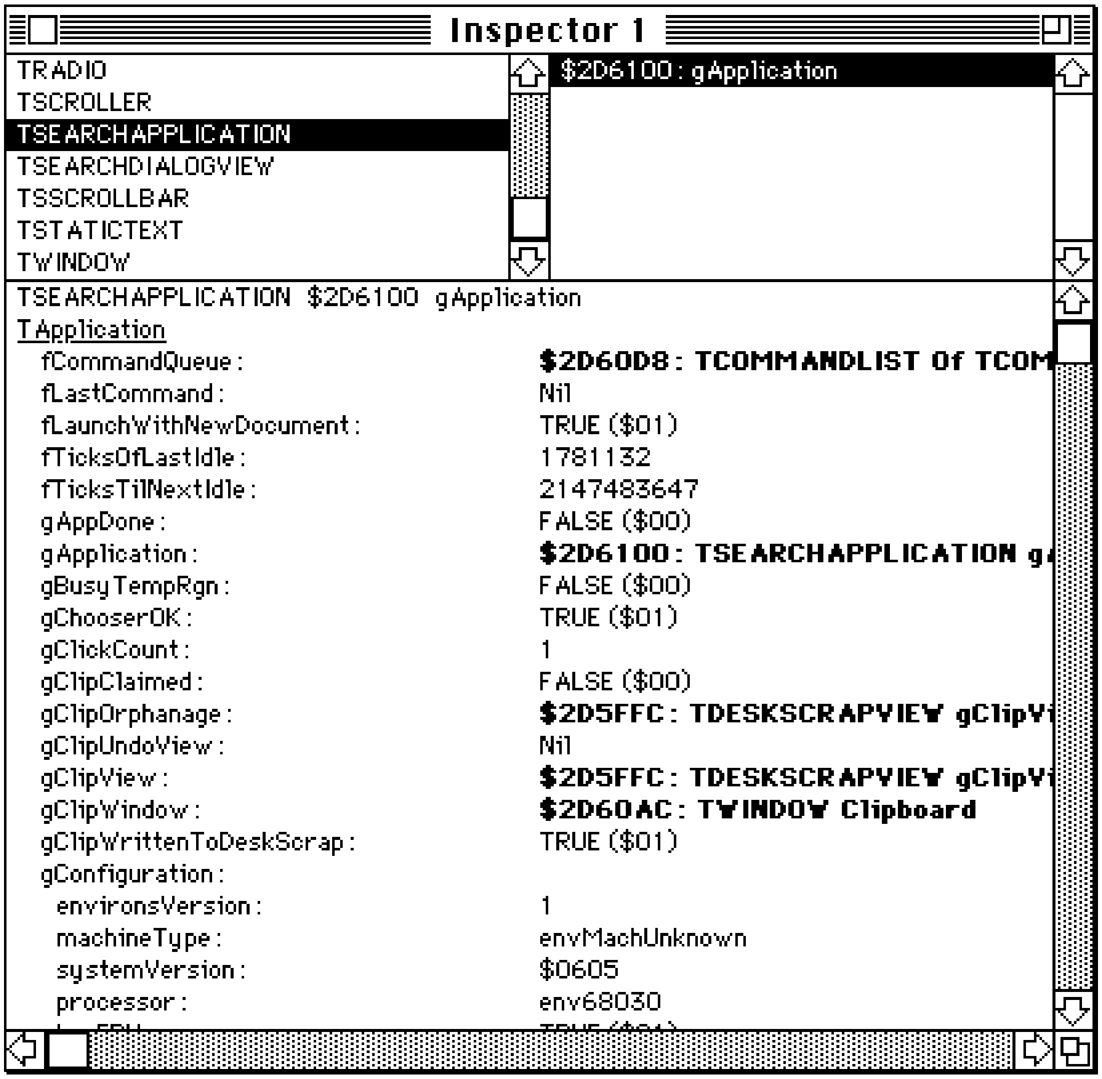
The first noticeable change is that some object fields are now boldfaced. Boldfaced fields are ones that bring up more information when you click on them. (You were always able to do this, but it was never apparent which fields were “active.”) Try it out. Bring up an Inspector window, and click on your application object. Then click on something like “gClipOrphanage,” which will be in boldface type. Doing so will remove the display of the application object, and replace it with a display of the fields for the gClipOrphanage. If we had held down the option key when we clicked on “gClipOrphanage,” a new Inspector window would have been automatically created, allowing us to view both the application and view objects at the same time.

If you change the size of your Inspector window, you’ll find that the column and upper pane widths now change the way you’d like them to. The upper panes now resize and relocate themselves so that they each take up half the width of the Inspector window. The two columns used to display the contents of an object’s fields do the same thing.

(There are many reasons for the implementation of a good idea. It was necessity that gave birth to this one. If you were to scroll down the list of TApplication fields, you would see two entries: “gDrawingPictScra…” and “gDrawingPictScra…”. These are two global variables whose names are identical for the number of characters that the Inspector allows by default. By being able to change the width of our columns, we can see that one is named “gDrawingPictScrap” and the other “gDrawingPictScrapView”.)

The instance variables that refer to owned or associated objects now contain a little more information. When the contents of those fields are displayed, the class name of that object is now displayed along with the object’s address and Inspector name. This should help you determine exactly what is what.

Finally, someone decided that the Inspector window needed a horizontal scrollbar. So what’s the big deal? This is MacApp!



```
Inspector 1

TRADIO                          $2D6100 : gApplication
TSCROLLER
TSEARCHAPPLICATION
TSEARCHDIALOGVIEW
TSSCROLLBAR
TSTATICTEXT
TWINDOW

TSEARCHAPPLICATION $2D6100 gApplication
TApplication
  fCommandQueue :                    $2D60D8 : TCOMMANDLIST Of TCOM
  fLastCommand :                     Nil
  fLaunchWithNewDocument :           TRUE ($01)
  fTicksOfLastIdle :                 1781132
  fTicksTilNextIdle :                2147483647
  gAppDone :                         FALSE ($00)
  gApplication :                     $2D6100 : TSEARCHAPPLICATION g
  gBusyTempRgn :                     FALSE ($00)
  gChooserOK :                       TRUE ($01)
  gClickCount :                      1
  gClipClaimed :                     FALSE ($00)
  gClipOrphanage :                   $2D5FFC : TDESKSCRAPVIEW gClipVi
  gClipUndoView :                    Nil
  gClipView :                        $2D5FFC : TDESKSCRAPVIEW gClipVi
  gClipWindow :                      $2D60AC : TWINDOW Clipboard
  gClipWrittenToDeskScrap :          TRUE ($01)
  gConfiguration :
   environsVersion :                 1
   machineType :                     envMachUnknown
   systemVersion :                   $0605
   processor :                       env68030
```

Figure 2 - Inspector Window

MacAppAlertFilter

Whenever MacApp shows a Dialog Manager Alert, it does so through a global bottleneck procedure called MacAppAlert. This procedure takes as its parameters an alertID number and a pointer to a procedure to be used as the Alert's event filter. If you don't supply a filter procedure of your own, then MacApp can use a default Alert filter. MacApp now has a global procedure that can be used as this default Alert filter. Called MacAppAlertFilter, it is responsible for two things.

First, it provides a convenient mapping of keyboard strokes to button controls (remember, since we are using a Dialog Manager Alert, the controls in the window are raw Control Manager controls; they are not managed by TControl objects). MacAppAlertFilter does this by taking the keystroke and comparing it to the first characters of the first three alert items. If there are any matches, then a mouse click is simulated on that control. All of this is done in a Script Manager compatible fashion.

]

The second feature MacAppAlertFilter provides is support of events belonging to windows other than the Alert itself. Any update or activate events that belong to other windows are passed on to those windows. In addition, idle events are passed on to the application so that idle time processing is not halted while the alert is on the screen.

We had originally intended to provide all of this to you by automatically installing MacAppAlertFilter as the default filter. However, we felt that the changes involved were too great, and wimped out. Just as it’s been in the past, there is no default Alert filter. If you would like to use MacAppAlertFilter as the default Alert filter, set the global variable “gMacAppAlertFilter.” This is a pointer to the default Alert filter that MacAppAlert should use if you don’t pass it one. MacApp sets gMacAppAlertFilter to NIL in its initialization routines, but you can set it to “@MacAppAlertFilter” in your I*Your*Application method.

Smart Zooming

A lot of work was done to TWindows to make them more intelligent about their positioning on the screen. In addition to ForceOnScreen and Center being made to work correctly, TWindow.Zoom was modified so that the window is zoomed to the screen it is currently on, rather than the main screen (the one with the menu bar). If the window spans multiple screens, it is zoomed to the one that contains most of the window’s contents. Check it out; it’s really neat.

Other Additions

There were four other additions made to the MacApp system that don’t involve the library. These include 3 units added to the sample programs, and a utility for cleaning up derezzed views.

One addition is UTabTEView, provided by Tom Dowdy at Apple. This unit defines a descendent of TTEView that supports TAB characters. The unit is kept in the DemoText folder, though it is not currently implemented by DemoText itself.

Another addition made to the samples is TMenu, by Larry Rosenstein. This descendent of TView provides support for custom menus. It handles installing itself into all the right hooks, and dispatches Menu Manager commands to the right methods. All you have to do is provide the code that draws the contents of the menus, handles the highlighting, and determines what menu item is selected. Use of this class is shown in the DrawShapes sample, where a simple patterns menu is implemented.

A minor eyesore in the Calc sample program was fixed by Larry Tesler, Vice President of Apple’s Advanced Technology Group. He provided a TSynchScroller class that allows two views to be scrolled at the same time. This means that you no longer have to see the main view of a Calc spreadsheet scroll slightly out of synch with its row and column headers.

Although these classes are provided with MacApp, I’d like to point out that they are mostly in a “read-only” state. They are there because they are useful and neat. However, they are not tested, nor are they supported as official Apple classes at this time. If you use them, you are on your own. Of course, if you do find bugs in them, we would like to hear about them.

Finally, we’ve included Curtis Faith’s script for cleaning up the output of Derez after it has processed a view resource. As you may have noticed, Derez’s output is rather sparse. Curtis’s script (called “CleanupDeRezzedViews”) strips out all the extra space, putting as many items as it comfortably can on one line. And frankly, as simple as this script is, it is very useful, as well as fun to watch. After derezzing a view resource of mine that had over 100 items in it, I sat back in my chair and applauded (it was 2:00 in the morning, so I could do that; no one else was around). CleanupDeRezzedViews gave me much more readable source code, which also happened to take up half the disk space as the original.

A small look at the future

Finally, with MacApp 2.0, you are given a small glimpse into the future of MacApp. There are many sections in the source code that are conditionally compiled based on the setting of the compiler variable qExperimentalAndUnsupported. Features that are currently conditioned out include offscreen double buffering for views, focus caching, and new descendants of TDynamicArray. These features will be implemented differently in the future but, this is a small hint of some coming attractions.