A C++ Application framework for Macintosh

created by Graham Cox

User Manual

Edition 3, January 2000
For framework version 2.1

Welcome to MacZoop!

This manual has been completely rewritten for the MacZoop 2.1 release. If you are familiar with earlier versions of MacZoop, you will notice a few changes in this version which help make MacZoop even easier to use and more powerful. If you are new to MacZoop, this manual will tell you all you need to know to get started writing Macintosh applications in next to no time. This manual completely replaces the old one.

MacZoop is aimed at beginners to Macintosh and C++ programming, but also at seasoned programmers who want to create applications that don't require a huge, heavyweight and bloated framework. If you are a beginner, the Getting Started and Tutorial chapters have been written especially to ease you into MacZoop concepts comfortably. If you have more experience, you may prefer to gloss over or skip these chapters and move straight on to the later chapters which explain how MacZoop is intended to work.

System requirements

MacZoop can be used to create applications for any Macintosh that runs System 7.0 or later. This includes both 680x0 and Power PC models. MacZoop is Appearance savvy if you have System 8 or later, and is designed so that the same source code can be used to create applications that will work across all versions of the OS since 7.0.

To use MacZoop, you require a compiler and development system. MacZoop is closely designed around Metrowerks' CodeWarrior, but can also be used with Apple's MPW development system. MacZoop is distributed as full, complete source code, so you can learn from the code supplied and even change it to suit your requirements if you have to. However, the basic framework is intended to work unchanged across any conceivable application, and should compile and run straight out of the box with no special difficulties. MacZoop will work with any version of CodeWarrior since DR4 or so, provided all C++ features such as exceptions and RTTI are supported. All CodeWarrior Pro versions are fine, as is CodeWarrior Lite.

Legal stuff

MacZoop is provided 'as is' and its fitness for any particular purpose is not warranted. It is the copyright and intellectual property of Graham Cox, and all rights are reserved. MacZoop is completely free, but a condition of its use is that any applications created with it clearly state in both their documentation and "about box" that they are a derivative work of MacZoop. The about box message should read:

created with MacZoop by Graham Cox, ©1994-2000 All Rights Reserved.

There are no other restrictions on the use of MacZoop for commercial or other purposes, provided the above condition is met. You may redistribute MacZoop in unmodified form, but the complete original package, including all documentation, must remain with it. If in doubt, please contact the author.

Feedback and support

Technical support is available for MacZoop by eMail. In addition, there are two mailing lists

which may be of interest to MacZoop users. The announcement list is used to get news about updates and important information about MacZoop, and there is also a discussion mailing list which is used to discuss technical and other issues regarding the framework. Most technical support can be obtained through this second list, but the author may be contacted directly if preferred.

Direct technical support- *MacZoopHelp@apptree.demon.co.uk*

Mailing lists:

Send an eMail to: *requests@lists.ranchero.com*

For the announcement list, the subject line should be “subscribe MacZoop-Announce”
For the discussion/support list, the subject line should be “subscribe MacZoop-discuss”
The message body should be blank.

Websites

MacZoop is available via a number of websites. From time to time, updates and fixes will be released and can be downloaded from these sites.

Main site:
http://www.kagi.com/tjriley/maczoop/default.html

Mirror sites:
http://www.wulfden.org/MacZoop/

This Manual

This manual is designed to teach you how to use MacZoop. It provides a tutorial and reference guide to all of the major features of the framework. What it cannot teach you is how to program in C++, nor is it a reference guide to Macintosh programming. While a quick introduction to Mac programming and object-oriented concepts is provided, you will need to refer to other books to get the most out of MacZoop, especially if you are a beginner.

To learn more about C++, I recommend

- *C++, The Complete Reference, 2nd Edition by Herbert Schildt, and published by Osborne/McGraw-Hill. (ISBN 0-07-882123-1)*

I also recommend the excellent online tutorials that comes with the CodeWarrior development system, namely

- *Learn C++ on the Mac*
- *Principles of Programming*

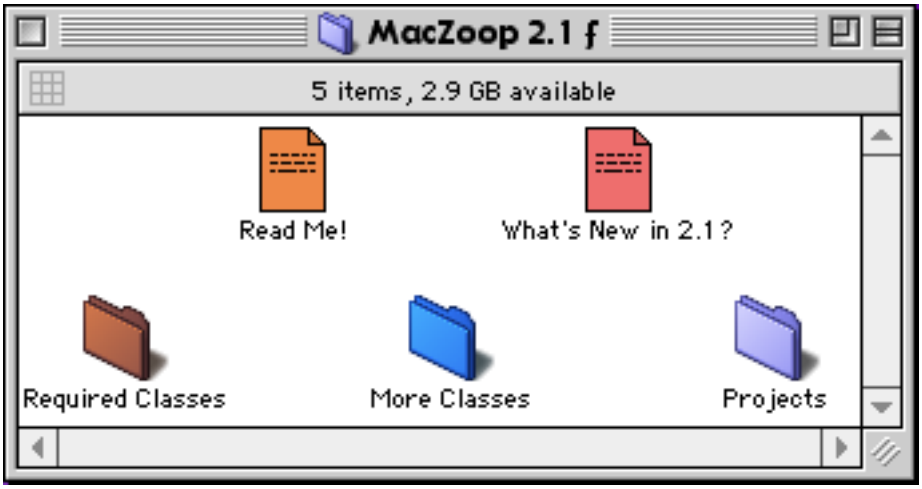
To program the Macintosh effectively, you need Apple’s technical documentation, “Inside Macintosh”, which is available for free from Apple’s website at:

http://developer.apple.com/techpubs/macos8/mac8.html

This introduction, and indeed this whole manual, approaches both the language and the platform from the perspective of the MacZoop Application framework, teaching you the principles you need to know to employ MacZoop for your own projects.

MacZoop

Before you begin, make sure you have all you need. The MacZoop package you downloaded or obtained on CD looks like this:



This manual may have been supplied with MacZoop (CDs), or downloaded separately to save time. In any case, you're reading it, so presumably you found it!

Required Classes contains the source code for the parts of MacZoop that every MacZoop project MUST have in order to work. More Classes contains source code for lots more useful code which can be added to projects as required. The Projects folder contains two pre-defined projects- a minimal MacZoop application, and a Demo application that shows many more of the framework's features. You will also be creating your own MacZoop projects in this folder.

The Read Me and What's New? documents contain useful or last-minute information about the distribution- check them out to make sure you're up to date.

Versions supported

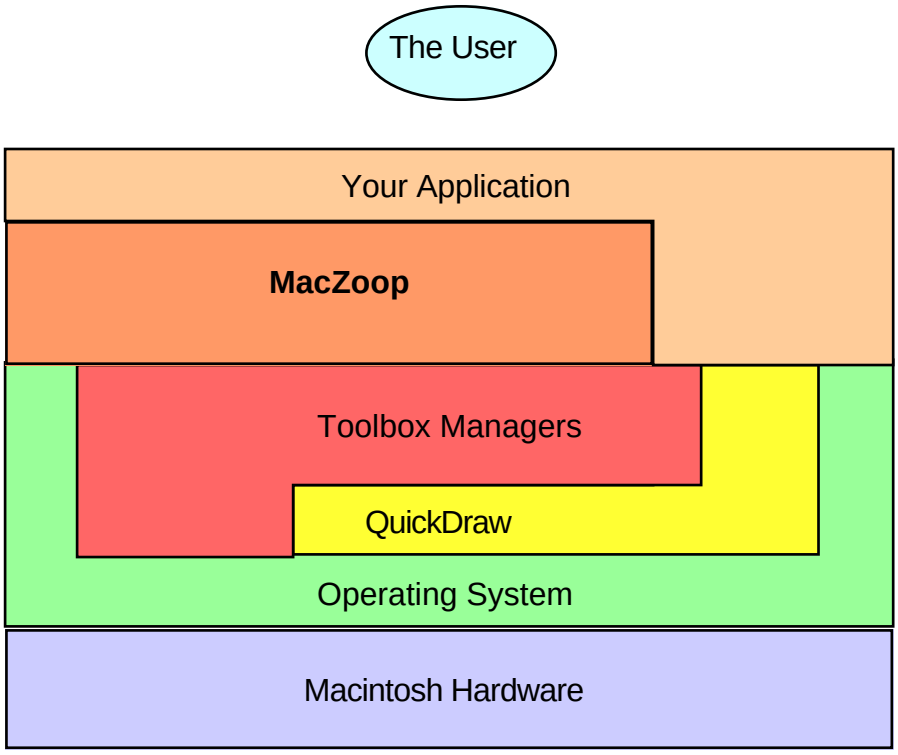
This manual is designed to document version 2.1 of the MacZoop framework code. It supercedes all previous manuals. As new versions of MacZoop are released, the manual may or may not be updated, depending on the nature of what has changed. Since the manual is a complex and time-consuming production, it is envisaged that frequent minor changes will not be made. Instead, this manual will stay current until further notice, and updated distributions will have additional "delta" documentation as needed. Be sure to read any documentation that is supplied with any future updates. Version 2.1 was released January, 2000.

Contents

Introduction to Macintosh Programming

The Macintosh is a modern computer embodying the principles of the Graphical User-Interface (GUI). Indeed it was the first mass-market computer to popularise this approach to program design. I assume that you are familiar enough with the basic operation of the Macintosh and your interest in reading this manual is to learn to program it. Among its many original features, the Mac pioneered the concept of a programmer's toolbox provided by the manufacturers to help the developer write their applications in a consistent manner without having to rewrite common GUI code every time. The Macintosh toolbox is a vast collection of routines to help you with many different parts of your programming design and implementation. However, in many ways this toolbox is quite low-level and provides only the tools to get a job done rather than any higher strategy or methodology for putting them together. This higher-level coding is left to the developer, and one of the purposes of a framework is to bridge the gap between the lower level toolbox and the application-specific code that makes your program interesting to you and the rest of the world. A framework such as MacZoop ties together many toolbox routines into a working whole which operates at a higher, more accessible level. This code provides a solid, fully debugged base on which you can build without wasting your efforts reimplementing the same basic code over and over again. Using MacZoop doesn't mean you don't have to learn the Mac toolbox, but it does avoid some of the need to rediscover the correct way to implement common things such as scrolling windows, dialog boxes, etc. These are completely solved problems, and should be treated as such. Life's too short to reimplement yet another main event loop!

This diagram illustrates MacZoop's place in the programmer's world:



Organisation of the Mac Toolbox

The Mac was originally designed to be programmed in Pascal. This legacy constrained its design in some ways, in that the toolbox is procedurally oriented, with related routines grouped into managers. For example, there is a Window Manager which is a collection of data structures and procedures relating to windows, a Control Manager for controls, a Dialog Manager for dialogs and so forth. Nowadays, this approach looks a little old-fashioned, and a modern object-oriented approach is far preferable. Once again, a framework comes to the rescue by providing an object-oriented Application Programming Interface (API) to the Macintosh managers. While you may occasionally need to call the toolbox directly, in many cases you can call a MacZoop object instead to achieve the same effect with less coding on your part. In some cases, MacZoop provides objects that have no direct relationship or correspondence to a toolbox manager, but embody vital standard functionality of use to the programmer nevertheless. For example ZApplication, the application object, has no toolbox counterpart, but its existence vastly simplifies writing a real application.

Where possible, MacZoop uses the available Macintosh managers to underpin its objects. The advantage of this is that MacZoop will usually automatically gain new features when the operating system is updated, and the toolbox code is extremely solid with millions of hours of development behind it. It also means MacZoop does not add too much weight to an application by reimplementing a lot of things available already. For example, many frameworks shun the Dialog Manager for doing dialogs- instead MacZoop builds on the dialog manager, enhancing it with such things as automatic radio-button grouping, listboxes, etc.

The routines and data structures provided by the toolbox are documented in “Inside Macintosh”, the ‘bible’ for all Mac programmers. You will need a copy of this (which now runs to many volumes) if you are going to be able to do any serious Macintosh programming, with or without MacZoop. Luckily “Inside Macintosh” is now available free online from Apple Computer. (http://www.apple.com/developers/) At a minimum I suggest you obtain the “Overview” and “Macintosh Toolbox Essentials” volumes.

QuickDraw™- Doing Graphics

One large and important part of the Mac toolbox that MacZoop does not provide an object-oriented API for is QuickDraw™. QuickDraw™ is a collection of graphics procedures for drawing lines, shapes, text, bitmaps, and for handling clipping to arbitrary regions, doing colour conversions, etc. You will use QuickDraw™ a lot in a typical application development, and with MacZoop this is no different. You use QuickDraw™ to draw in MacZoop windows just as you would a “native” window. MacZoop also uses numerous QuickDraw™ data types within itself, such as Rects, Points, Regions, and so on. To fully understand QuickDraw, you’ll need “Inside Macintosh: Imaging with QuickDraw”.

Lightning tour of QuickDraw™

The basic data type in QuickDraw is the Point, representing a single coordinate on the graphics plane. The graphics plane is a logical space from -32,767 to +32,768 pixels in both horizontal and vertical dimensions. The origin is at 0,0 and increasing values go to the right and down.

To provide a drawing environment on the graphics plane, QuickDraw defines the GrafPort data type. This provides a structure for the coordinate plane, and associates it with a section of

memory (usually video RAM) into which graphics primitives are rasterised. The GrafPort also defines a pen for drawing, consisting of a pattern, colour, width and height, drawing mode and so on. Text characteristics are also set up here.

Drawing is always done in the current port. A window is built on a GrafPort, and drawing in a window requires that that window is made the current GrafPort at the time the drawing is done. This should not be confused with the active window- there is no relationship between the active window and the current port- it is perfectly possible (indeed necessary) to draw into inactive windows.

A rectangle (Rect) is a basic data type that is widely used. This consists of two points, defining the top, left and bottom, right of the rectangle. As well as being able to draw rectangular frames, fills, etc. QuickDraw uses rectangles as abstract structures in many places, for example defining clipping, etc.

The Region is an important structure which represents an arbitrary (and not necessarily con-joined) collection of pixels. Regions are widely used for clipping to arbitrary shapes, and they are very powerful in that they can be added, subtracted and exclusive or'ed, as well as framed and filled. Most drawing involves regions at some point.

For colour drawing, QuickDraw utilises a 48-bit RGB colour record (RGBColor). Every GrafPort can have the foreground and background colours set to an arbitrary RGBColor, and subsequent drawing is done in that colour, reduced to the colour resolution of the target device. Usually you work with 48-bit colour values and allow QuickDraw to perform the appropriate reductions.

Though you work with QuickDraw directly in MacZoop, you do not need to be all that concerned with setting the current port, or offsetting the origin to allow for scrolling, etc. MacZoop expects you to draw your window contents by overriding a method (DrawContent()) that is called after all of the standard QuickDraw set up is already done.

Events

In order to provide responsive, interactive applications, the Macintosh programming model incorporates the concept of events. An event is something arising from the users actions, for example typing on the keyboard, or clicking the mouse. The typical application sits in a loop waiting to receive an event, then goes off and processes it. When done, it resumes waiting for the next event. MacZoop is built around this basic model, but unlike programming the Mac directly, you generally won't see these events in themselves, but more likely will implement particular methods to handle them where the standard handling provided by the framework is not quite what you want. The standard responses to all events provided by the framework are designed to implement the normal, expected conventional behaviour for a Mac application- for example, clicking in a background window selects it, or choosing "New" from the Edit menu makes a new, untitled document. The standard behaviours are defined in the "Human Interface Guidelines", a book published by Apple and available online from their website. Once again, by implementing the guidelines, MacZoop relieves you of the need to worry about them. Build your code with MacZoop and you'll automatically have a well-behaved Mac application that works the way your users have come to expect.

Windows

Generally, all user interaction with an application takes place through a window. This is just an area of the computer screen that provides some sort of view of an internal data model. In a word-processing application, for example, the data model is a list of character codes, and the view of that data is an array of character shapes corresponding to the character codes. In general, good application design hides the data model from the user in such a way that to them it feels as though the data is being manipulated directly by them. This is the principle behind most modern computer programs- to the user, a Finder icon is a file for example. Since the window is fundamental in providing a generic way to view data, naturally much programming effort revolves around it. MacZoop makes creating and manipulating windows and the data models they represent very straightforward- much more so than the 'native' routines in the Mac toolbox.

Only one window can be active at once. This is called the active window and will usually be the frontmost one. Note that floating 'utility' windows are never considered to be the active window, even if they appear to be in front of the window that is. For this reason, it is important that such utility windows have a distinct visual appearance, and are generally kept small.

Within the active window, some other entity may have the user's attention. This is called the selection, and may take many different forms. In a text document for example, the selection is indicated by a highlighted area of text in a contrasting colour. In an icon view, the selection is indicated by a darkened icon with the text inverted.

Within MacZoop, window activation is largely taken care of for you, as is moving, zooming, closing and resizing them. Implementing a data model and providing a view to it is usually all you'll be concerned with, and this can be done with only a handful of overridden methods. MacZoop will also handle all the chores associated with implementing a scrolling view in a window if you need one, as well as drag and drop, and implementing text and picture-based data models.

As mentioned, MacZoop provides a floating window capability, and these windows are automatically set up by using particular window definition codes. Otherwise, handling the content of these windows is no different to any other window. MacZoop builds on the Macintosh window manager to get its work done. You will rarely if ever need to call this part of the Mac toolbox when using MacZoop.

Commands and the Menu Bar

The Menu Bar is the strip along the top of the screen containing the pull-down menus that trigger particular functions in an application. The array of options in the menus give the user a useful "road-map" of your application and its importance in orienting the unfamiliar user to your app should not be underestimated. MacZoop makes it easy to handle menu choices by abstracting the menu items into the concept of commands. Instead of propagating the actual menu item through the program, a command is sent which stands for that item. By abstracting commands in this way, your application code is simplified while creating a simple and easy way to maintain the state of the menubar, so important for the user experience. Another advantage it confers on the application designer is that the menu items are only bound to the commands at runtime, thus allowing the layout of the menus to be changed at will without any recoding. MacZoop also handles the standard commands New, Open, Close, Save, Save As, Revert, Page Setup, Print, Quit, Undo, Cut, Copy, Paste, Clear, Select All and the "About" command in standard ways that

you’ll rarely need to override.

As your knowledge increases with practice, you’ll find MacZoop provides simple objects to do all sorts of chores for you- files, progress bars, all kinds of dialog boxes, offscreen drawing and so on are all provided for you. Often these objects can be employed as they come or subclassed to add functionality useful to you. Doing this requires that you know a fair bit about object-oriented programming, which brings us onto the next section...

Programming in C++

MacZoop is written in C++ and it is assumed that you will be writing your application in C++ too. This section will introduce you to a few key concepts of object-oriented programming with C++- enough so that you can get on with using MacZoop to build your next Mac application. I assume that you already have a working knowledge of C. C++ is effectively a superset of C though it is a different language. I do not intend to get into the nuts and bolts of C syntax here- that's admirably covered elsewhere. What I do want to get into is to explain why object-orientation is a good way to program, and how that is done in practice within the context of the MacZoop framework.

The main reasons for using object-oriented programming (OOP) are as follows:

- Efficient code reuse
- Smaller API for a given functionality
- Encapsulation of data and code
- Easier to manage large projects

The last point follows from the first three and is possibly the main motivation for using C++ in industry. C++ code is more maintainable, reusable and flexible and that means it costs less. But let's look at the first three points in more detail.

Efficient code reuse

A well designed object can be used again and again, whether within a single program or in several different ones. For example, MacZoop has a window object, ZWindow, that is reused whenever a window is displayed, no matter what the window itself is a view of. Every single window shares a lot of common code, which makes it use much less run-time memory, and makes it faster to write, and with luck, less buggy.

A smaller API

This results from two features of C++. One is the object-orientation itself- you can send ANY window an activate message, for example, by calling the window's Activate() method. The actual implementation for Activate() may differ from window to window, but you only need to know what it's called for any window. Secondly, C++ allows polymorphism, which is the ability to have similarly named methods with different parameter types. The compiler will pick the correct one according to the types you pass it. For example, you might have a control object that can store a value, and it has a method called SetValue(). If you define a version of this for an integer, a float, a string or any other data type, you still only need to learn one call- SetValue() and let the compiler figure out which one to use. In the C world, you would have needed a SetValueInt(), SetValueFloat(), SetValueString(), and so forth to do this, and you would have had to learn of each one separately.

Encapsulation of data and code

This third point is perhaps the most striking difference from the C world, and one of the most useful features of object orientation. In a classic structured language such as Pascal or C, data and code exist as separate entities. You devise a data structure, then a set of routines that can operate on this data, often passed as parameters, or stored in variables. This approach exposes the

data to you as a programmer, often when entirely unnecessary. For example, you might have some kind of record that represents the internal state of some data model. If you want to extend that record, everything that has ever seen it or might use it in the future has to be recompiled. If instead that data model is embodied in an object, it can be extended and modified until the cows come home, and no other code outside of the object need be any the wiser.

What is an object?

An object can be visualised in many ways. Here are some:

- A binding of data with the code that operates on it
- An opaque data model
- A single element of functionality in a program

Getting a grasp of what an object is is the hardest part of learning the object-oriented approach to programming. All three definitions above are right, but none tell the whole story. To the C programmer, an object is like a data structure that contains its own code. At a more abstract level, it is like a "black box" that performs operations in response to messages from the outside world, doing goodness-knows-what internally but doing it in a consistent and predictable fashion. The point about being a single functional element is often correct, but may not be always the case. Functionality as a whole may be implemented by several objects in cooperation, or a single object may have a number of functionalities. Often the choice of which objects to place particular functionalities in is arbitrary, but good object-oriented design is usually more of an art than a science. Here are some of the objects in MacZoop:

- ZApplication - the application itself
- ZWindow - any window or dialog box
- ZFile - any data file
- ZCommander - an object that can respond to a command
- ZMenuBar - the menubar management object

The Macintosh lends itself to an object-oriented design methodology. Whether they are implemented as objects in fact I do not know, but conceptually the following things in the Macintosh world are objects:

- A file, displayed as an icon in the Finder
- The Trash Can
- The Hard Disk icon
- Any window
- Any application, control panel or extension

When using the Mac, the usual paradigm is: "Hey You! Do this...". In practice this is achieved by making a selection, then choosing a command, either explicitly from a menu, or implicitly by a drag, for example. Next time you open an application, it'll go something like: Click on application "MacWrite" ("Hey You, MacWrite!..."), go to the File menu and choose "Open" ("Launch yourself!"). Of course you would use a double-click as a short-cut, but you get the idea.

Object-oriented programming is much the same conceptually, but at the coding level. You identify the object you want to use ("Hey You!..."), then send it a message ("Do this..."). In C++, this is done using the pointer operator, e.g:

```
myWindow->Activate();
thatDialogBox->Close();
thisChocolateBox->Open();
heyYou->DoThis();
```

In the above example, the objects <myWindow>, <thatDialogBox>, <thisChocolateBox> and <heyYou> are all object variables that you have to hand. An actual existing object variable is known as an instance of that object. Often you will have many instances of the same object class, but they are all different objects- they just happen to encapsulate the same functionality.

Good object design embodies the idea of autonomy. Where possible, the object should not require too much outside of itself to get its job done. It can be taken from one context and employed in another unchanged. This ideal permits code reuse between projects as well as within a single project. In practice the ideal is rarely achieved, but a good designer will strive for it nevertheless.

Making Objects

There are two ways to make objects, or rather two places- in the heap or on the stack. Stack objects are temporary objects that exist only as long as the function or method they are declared within remains in scope. More on this later; for now we will concern ourselves with heap objects which have a more permanent existence since they have to be deliberately brought into being and deleted again when they are no longer needed. C++ provides two keywords to do this: new and delete.

Suppose we have a class called ZChocolateBox. Here's how we would make a new instance of it in the heap:

```
ZChocolateBox*    thisChocolateBox;
thisChocolateBox = new ZChocolateBox();
```

This example assumes that ZChocolateBox takes no parameters to its constructor. The constructor is a method of the object that is called automatically as soon as the memory for the instance is allocated. More on this in a moment. Note that the variable <thisChocolateBox> is declared as a pointer type using the * operator. This is necessary for all objects created in the heap.

Once we have finished with <thisChocolateBox> (perhaps we've scoffed the lot and want to discard the wrapper), the delete operator is used:

```
delete thisChocolateBox;
```

Defining Objects- Classes

So you've got an idea for an object- how do you actually define it? This is done using the class

keyword. This is similar to using the keyword struct in C (in fact the two are effectively synonymous in C++)

```
class ZChocolateBox
{
public:
      short       numberOfChoccies;
      short       numberOfLayers;

      ZChocolateBox();
      virtual ~ZChocolateBox();

      virtual void      Open();
      virtual void      EatChoccie( short chocIndex );
};
```

Now, in C, you might have designed your program like this:

```
typedef struct
{
      short       numberOfChoccies;
      short       numberOfLayers;
}
ChocBoxRecord;

void        NewChocBox( ChocBoxRecord* aBox );
void        DisposeChocBox( ChocBoxRecord* aBox );
void        OpenChocBox( ChocBoxRecord* aBox );
void        EatChoccie( ChocBoxRecord* aBox, short chocIndex );
```

You see how much neater the C++ technique is? The class keyword allows you to define your functions (they're called methods if they're part of a class) as if they were fields of the structure itself. All that tedious parameter passing of some arbitrary data type is avoided, since the methods of a class can always see the data fields (they're called members) without being told where to look explicitly. In addition, every class has a constructor and a destructor. These are methods that are called automatically by new and delete and are used to initialise the object and to clean it up just before it is deallocated. In C++, the convention is that the constructor has the same name as the class, and the destructor has the same name but preceded by a tilde (~) symbol. Though just a convention, most compilers insist that you follow it, and anyway it's good practice. Because these methods are guaranteed to be called at the relevant times, they are a good place to put your set up and clean up code. Note that the constructor must exist, but the destructor is optional. Almost needless to say, but once an object has been destructed, you must not make any other calls to it- any object references you may have around become stale after a call to delete.

Inheritance

The C example above, apart from its basic unwieldiness, has another basic flaw. If you wanted to make something that was almost the same as a chocolate box, but needed a different bit of code

to implement EatChoccie(), you would have to go to a lot of trouble to implement it. You have to have a way to distinguish between the type of chocolate box, then modify your existing code to check for the type and either call the old or the new routine to do the right thing. For example:

```
typedef enum
{
	blackMagic,
	milkTray,
	allGold
}
ChocBoxType;


void	DispatchEatChoccie( ChocBoxType cType,
				ChocBoxRecord* aBox,
				short chocIndex )
{
	switch( cType )
	{
		case blackMagic:
			EatBlackMagicChoccie( aBox, chocIndex );
			break;

		case milkTray:
			EatMilkTrayChoccie( aBox, chocIndex );
			break;

		case allGold:
			EatAllGoldChoccie( aBox, chocIndex );
			break;
	}
}
```

Note that this technique is not only tedious, but does not lend itself well to code reuse, since to extend the types of chocolate boxes you can deal with, all the basic “outside world” code has to be revised. C++ solves this problem using inheritance. With inheritance, you derive a new class from an existing one, and reimplement those methods that need to change in the new class. The code that calls the object does not know that the object is not the same as the original one, and so does not need to be revised for the new case. For example:

```
class ZBlackMagicBox	: public ZChocolateBox
{
public:
	ZBlackMagicBox();

	virtual void	EatChoccie( short chocIndex );
};
```

Now it doesn’t matter whether <thisChocolateBox> is a generic ZChocolateBox, or a ZBlackMagicBox, you can call the EatChoccie() method and it will do the right thing. No code changed outside of the new object itself, and there only the different parts had to be additionally coded. (Open() for example, is not recoded for this object). The new object inherits the behaviour of the one it is based on in all respects except where it has deliberately overridden it.

In the class definitions, the use of the word virtual ahead of the method prototype indicates that it would be possible for a subclass of this object to override that method. If the word virtual is not there, the method can't be overridden. In MacZoop, most methods are overridable which allows you great flexibility to customise your objects.

Note that unlike C, in C++, all methods MUST have a prototype. What was very good practice in C is now compulsory.

Storage Classes

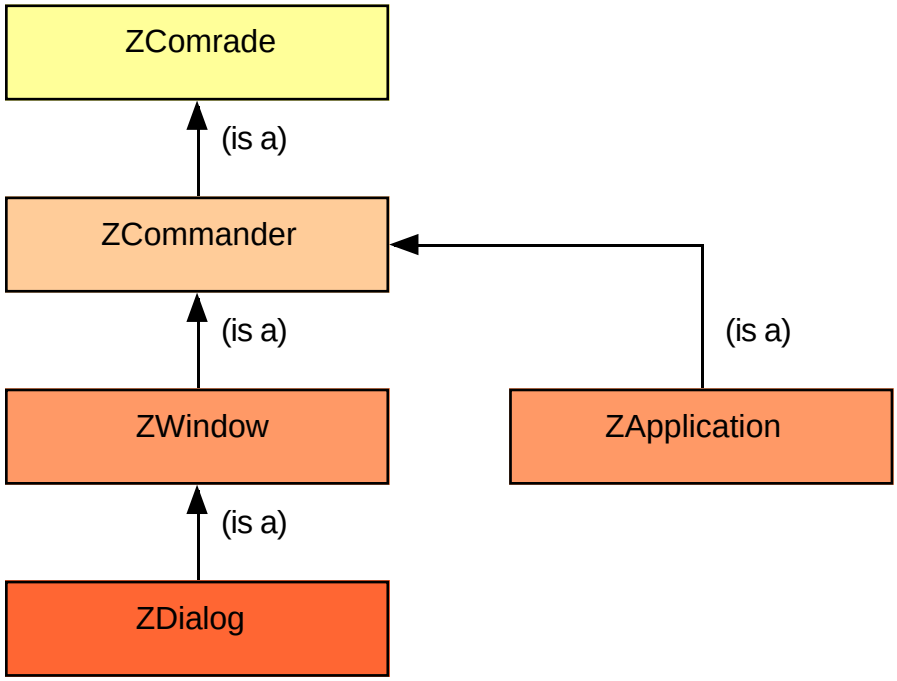
Now what's all that "public" nonsense? This keyword defines what is called the "storage class" for the data members and methods that follow it. There are two others, "protected" and "private". Public means that the data members and methods can be accessed from anywhere in the entire program. While often a good idea for methods, you'll normally want to prevent abuse of your object by protecting the data members from external changes. The keyword "protected" does just this- your class and anything that inherits from it can access the data members directly, but anyone else cannot. Protected is the most useful storage class for data members, and many methods too. "Private" is very similar, but additionally prevents access from derived classes as well the outside world. Not so often used. A better way to define ZChocolateBox would be:

```
class ZChocolateBox
{
protected:
      short       numberOfChoccies;
      short       numberOfLayers;

public:
      ZChocolateBox();
      virtual ~ZChocolateBox();

      virtual void      Open();
      virtual void      EatChoccie( short chocIndex );
};
```

Here, <numberOfChocolates> and <numberOfLayers> are accessible to the methods of the class and its subclasses, but no-one else. The other methods are accessible from anywhere. Note that constructors and destructors should always be public.

Class Hierarchies

Let's look at a concrete example from MacZoop:



This chart is an example of a class hierarchy. CodeWarrior has the ability to generate these charts from your source code automatically, and it is very instructive, especially when learning a new framework, to study the chart to see how things relate to each other. In this example, ZDialog inherits from ZWindow which inherits from ZCommander which inherits from ZComrade. This means that ZDialog is a ZComrade, as well as the other things in between. The functionality in ZDialog is the sum of the functionality of the classes between itself and its root class. ZApplication is also a ZCommander and a ZComrade, so it has the functionality of those two as well as its own, but does not inherit the code brought in by ZWindow or ZDialog. ZApplication may well have to access ZWindow objects, but it isn't one itself, which would of course be absurd.

Now as it happens ZCommander defines a method called HandleCommand(), which is very useful and frequently called. To get an object to handle a command, you only need to know that it is a ZCommander. The actual reality that it may be a ZWindow or a ZDialog or a ZApplication object is of no importance- you can think of them as being the same thing at the ZCommander level, e.g:

```
ZWindow*          aWindow;
ZDialog*          aDialog;
ZApplication*     theApplication;

aWindow->HandleCommand( kSomeCmd );
aDialog->HandleCommand( kSomeCmd );
theApplication->HandleCommand( kSomeCmd );
```

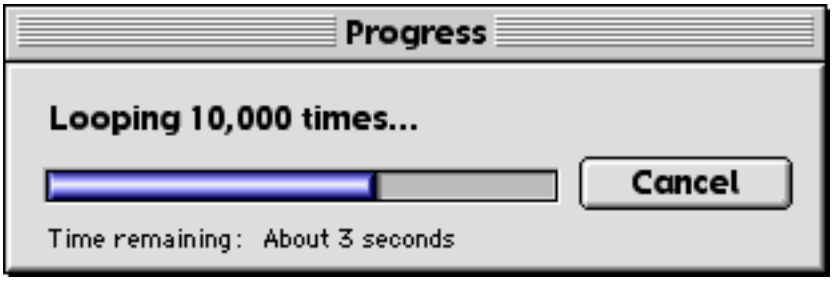
Whether or not ZWindow or the other objects override HandleCommand() to do something

useful is neither here nor there to the caller- it is not interested in what is done, only that it is done, somewhere or other. Hopefully you can begin to see why this is so supremely powerful. It allows frameworks to exist and be deployed in a myriad of diverse applications with no changes at all to the basic code. Instead of having to design a window object from scratch, you inherit from ZWindow and get all the standard behaviour, then override the few bits that need to be different- perhaps the content drawing method for example.

Your programming effort is then concentrated on the differences between the standard behaviour and what makes your application special and useful, and you don't waste time reimplementing all that tedious stuff just to make a window appear on the screen, or give it scrollbars, and so forth.

Stack Objects

Sometimes you want an object to exist for a short time within a well-defined scope to carry out some task for you. Stack Objects can be very useful here. Stack Objects do not need new or delete to make and destroy them, they'll be made on the stack automatically just like any other variable. As with heap objects, the constructor and destructor methods are automatically called when the object comes into and out of scope. Lets look at an example. MacZoop has a class called ZProgress which implements a progress dialog box, as shown below.



While it is perfectly feasible to create this as a heap object, for many uses a stack object can be easier to use. Typically, ZProgress is used to monitor a lengthy loop function and provide feedback to the user. As a programmer you want to concentrate on programming your loop, not the progress bar, so you want the overhead of this to be small. Here's an example of a loop function without progress feedback:

```
void  MyObject::Loop( long loopMax )
{
      long        i;

      for( i = 0; i < loopMax; i++ )
      {
            // execute a series of statements to do something useful here


      }
}
```

Adding a ZProgress stack object:

```
void  MyObject::Loop( long loopMax )
{
      ZProgress   zp( this, loopMax );
      long        i;

      for( i = 0; i < loopMax; i++ )
      {
            // execute a series of statements to do something useful here

            zp.InformProgress( i );
      }
}
```

Adding progress support adds two lines of code. Not so clear is the fact that you've also added the ability to cancel the operation automatically, as well as made it cooperate with other processes on your computer. So what's going on here?

The object itself is instantiated by declaring it on the stack as the variable zp of type ZProgress. Note the absence of the pointer(*) operator in this case. The constructor parameters are passed directly as part of the variable declaration, possibly the most unusual sight for a C programmer. The method of ZProgress called InformProgress() is called using the dot operator not the pointer operator, just as if it were the field of a struct (which it is). This method returns a boolean result indicating whether the user hit cancel or not, and if they did, an exception is thrown to abort the loop. The progress dialog object is destructed when the function enclosing it goes out of scope, so that will occur whether the loop terminates normally or abnormally due to the exception.

Stack objects are very useful in this sort of situation where you want a helper object that will clean up after itself without your intervention. They are also useful for saving state data for the duration of the enclosing function- you can save state data in the constructor and restore it in the destructor, automatically implementing a push and pop operation of almost any complexity. MacZoop provides a class, ZGrafState, for saving the state of the current GrafPort when instantiated and restoring it when destroyed, so providing a useful push and pop of the graphics environment around drawing calls.

Implementing Objects

So far we have talked about declaring and instantiating objects, but not actually coding them. In fact this is the least difficult concept to grasp, especially if you're used to C. Methods are written just like a C function, with the one difference that the method name is preceded by the class name and two colons. To go back to our chocolate box class example, we might implement EatChoccie() thus:

```
#include    “ZChocolateBox.h”

void  ZChocolateBox::EatChoccie( short chocIndex )
{
      if ( chocIndex >= 0 && chocIndex < numberOfChocolates )
      {
            ZChocolate*       theChoc;

            FailNIL( theChoc = RemoveChoccieFromBox( chocIndex ));

            theChoc->RemoveWrapper();
            Swallow( theChoc );

            delete theChoc;
      }
}
```

This example assumes we’ve added some methods to ZChocolateBox called RemoveChoccieFromBox() and Swallow()- a not terribly likely scenario, but stick with it, it gets better. We also assume that we have a class classed ZChocolate which implements an individual chocolate object- that might be subclassed for different flavours- ZStrawberryCream, or ZTruffle for example- whatever, we don’t care since we’re just picking out a chocolate by some index value.

First the index is checked against the knowledge we have of the number of chocolates- notice that we can access our own data members without having to care where they are stored. If the index is valid, we know we can safely get the chocolate from that position in the box, as done by RemoveChoccieFromBox(). This returns the object reference of the chocolate at that position (somehow- we do not concern ourselves with ‘how’ here), or NULL if the chocolate was taken earlier, in which case the FailNIL will throw an exception and abort the method.

Once we have the <theChoc> object reference, we call its RemoveWrapper() method. Now we don’t care how this is implemented- ZStrawberryCream might have a twist-off wrapper, whereas ZTruffle might come like a little cupcake- no matter, we ask the chocolate object to RemoveWrapper() in whatever way is right for that chocolate. Then we call Swallow(), passing the chocolate to swallow. Once swallowed, it’s gone so we delete it.

If you have a method that overrides one in the superclass, you still may want to execute that one before or after your own code runs. This is simply a matter of calling the inherited method. Usefully, there is an “inherited” keyword that identifies your immediate superclass, but unfortunately this keyword is not part of the draft ANSI spec for C++, so the compiler friendly way to do it is to explicit refer to your superclass by name, e.g:

nice, but non-ANSI way:

```
inherited::SomeMethod();
```

more compatible, but more ugly way:

```
MySuperClass::SomeMethod();
```

Constructors and Destructors

Constructors and Destructors are implemented much like any other method, but there's one difference you need to watch out for with constructors, and that's if you derive from another class. Let's look at the constructors for ZChocolateBox and ZBlackMagicBox to see how it's done.

```
ZChocolateBox::ZChocolateBox()
{
	numberOfChocolates = 16;
	numberOfLayers = 1;
}
```

The constructor initialises the data members to their proper values. Normally, it is good practice to initialise every data member in your object to some initial value, even if you believe that your object allocator clears memory to zero. Some implementations don't and uninitialised data can lead to obscure bugs that are hard to find. This is especially important with pointers. Get into the habit of methodically clearing or setting all your data members in your constructor- it adds virtually nothing to your execution time and makes your code stabler. Do it!

Now, it's a fact of C++ that constructors are not virtual methods by definition, and thus you can't override the constructor of your superclass. However, you need to make sure that the superclasses' constructors are called nevertheless, so C++ provides a special syntax for this, e.g:

```
ZBlackMagicBox::ZBlackMagicBox()
	: ZChocolateBox()
{
	numberOfLayers = 2;
}
```

The colon operator followed by the name of the superclass's constructor ensures that the superclass is constructed, then the rest of your constructor is executed. The constructor for ZChocolateBox sets number of chocolates to 16, and layers to 1, but the number of layers is subsequently amended to 2 in ZBlackMagicBox's constructor.

Destructors, by contrast, are generally virtual and the objects in a class hierarchy are automatically destructed in order from highest to lowest without you needing to pull any tricks of the kind you do with constructors. You just need to remember to declare the destructor of the root class of any hierarchy as virtual.

Another constructor pitfall

Because constructors can't be overridden, and they are called from lowest to highest in the class hierarchy when building an object, it means that if your constructor calls another class method (perfectly legitimate), it will call the one defined for the object at this level only, even if that method is overridden further up. This can become important if you need overridable initialisation methods. In this case, you need to split out the initialisation method from the constructor, and call it from the outside world instead of the constructor itself. In other words, the object must be fully constructed for an overridden method to work, but when the constructor is called, the object may not be fully constructed if another object derives from it.

A major example of this in MacZoop is ZWindow. The constructor for ZWindow will initialise some variables, but the window itself is built by a method called InitZWindow(). Classes deriving from ZWindow, for example ZDialog, may need to implement InitZWindow() in an entirely different way- this is true in this case since a ZDialog is built from 'DLOG' and 'DITL' resources whereas a plain window is built from a 'WIND' resource. Thus the constructors for these objects do not call InitZWindow(), but rely on the outside word to call it. You MUST call it to actually make a functional window- if you don't subsequent attempts to call the window object are likely to crash. Where this is necessary in the MacZoop framework, the documentation will clearly indicate this.

Example: Making a Window in MacZoop:

```
void  MyApplication::MakeNewWindow()
{
      FailNIL( mostRecent = new ZWindow( this, myWindowID ));

      mostRecent->InitZWindow();
}
```

Example: Making a Dialog:

```
void  MyApplication::OpenDialog( short id )
{
      ZDialog*    aDialog;

      FailNIL( aDialog = new ZDialog( this, id ));

      aDialog->InitZWindow();
      aDialog->Select();
}
```

Some other C++ stuff

If you need an object reference to yourself (a pointer, that is), you can use the keyword this. Normally you can call your own methods without any special scope resolution, but <this> can be handy to make ambiguous calls clear. It is also often used to pass a reference to yourself on to another object.

Sometimes a method name in your object masks an identically named function outside of your object, perhaps a toolbox call. If you invoke the name directly, your own method of that name will be called. If what you really wanted was the external function of the same name, you can force the compiler to resolve to a global scope using two colons, e.g:

```
SetCursor( aCursor );   // my method
::SetCursor( aCursor ); // the toolbox routine of that name
```

In general it's a good idea to avoid using the same names as toolbox calls for your methods.

In C++, method and function names are mangled with their parameter types for the purposes of identification to the compiler and linker. This is what permits polymorphism, but can cause problems, particularly when overriding a method. To override a method, both the name and parameter types must match exactly, or the method will be considered to be something different, not an override.

In this example, a class declares a method called DoSomething() with a const short parameter. A subclass wants to override DoSomething() but the programmer inadvertently omitted the 'const' part of the parameter list. The compiler treats the two methods as entirely different things and naturally this can lead to all sorts of difficult to spot bugs.

```
virtual void      DoSomething( const short aParam );
virtual void      DoSomething( short aParam);   // will not override the
                                                // above
```

C++ permits multiple inheritance. Though not widely used in MacZoop, it can be useful. It allows a class to derive from more than one superclass, and blend the functionality of the two. The syntax is a logical(ish) extension of the single inheritance case:

```
class MyCombi     : public ClassA, public ClassB
{
public:
      MyCombi();
      // etc.
};

// constructor:

MyCombi::MyCombi()
      : ClassA(), ClassB()
{
}
```

Organising your files

As in C, C++ compilers work with headers and implementation (code) files. In general, it is good practice to confine one class to one header file and one implementation file. The header should include all of the constants and definitions used by the class so that it can be moved to another project easily. The implementation should include the minimum necessary to allow it to compile.

The conventions for headers are, as in C, that they and in .h, and for implementation files to and in .cp, .cpp, or .c++. MacZoop uses the first of these throughout.

The main() function

As in C, every C++ program must have a main() function. In MacZoop, the purpose of main() is to instantiate an object of type ZApplication (or a subclass of it) and tell it to run. In MacZoop, you do not usually write your own mani() function. Instead, the ProjectSettings file sets up the application class that is made by the standard one. Therefore, you can pretty much forget about it.

Well, that's about it for this quick tour of C++ programming. I hope you'll now be able to work your way around the main part of the MacZoop manual without too much trouble, and more importantly, get programming!

Getting Started with MacZoop

This is the part that you probably turned to first- how the hell do I get this thing to run?! Well, the good news is that getting MacZoop to run should be fairly painless. The bad news is that everyone seems to have a slightly different set-up, so there may be kinks and bumps along the way that were not anticipated. This chapter will help you get things running, and understand what's going on so that you can deal with any little problems that are thrown in your path!

CodeWarrior

CodeWarrior is everybody's favourite development environment on the Mac, and with good reason. It's easy to use, powerful, and creates excellent quality code. The downside is that there are many different versions of CodeWarrior out there. This reflects the (sensible) policy of Metrowerks in releasing continuous upgrades to CodeWarrior over time, but has created a veritable Tower of Babel in terms of file formats and features. Many newcomers to Mac programming have CodeWarrior Lite, whereas seasoned professionals probably have the latest professional edition, version 5 (at the time of writing). Each version of CodeWarrior Professional has altered the file format of the project file, with limited backward support (but no forward support- CW Pro 1 for example, can make nothing of CW Pro 5 files). The problem is even worse, since CodeWarrior doesn't start with pro 1, it goes back much further. In fact Pro 1 is really more like CodeWarrior 2. However, before you give up in horror, be assured, all is far from lost. The really good news is that unless you have a REALLY ancient version of CodeWarrior (say DR4 or earlier), MacZoop will compile and run just fine.

That's such an important point, I'll say it again:

MacZoop works with ANY version of CodeWarrior

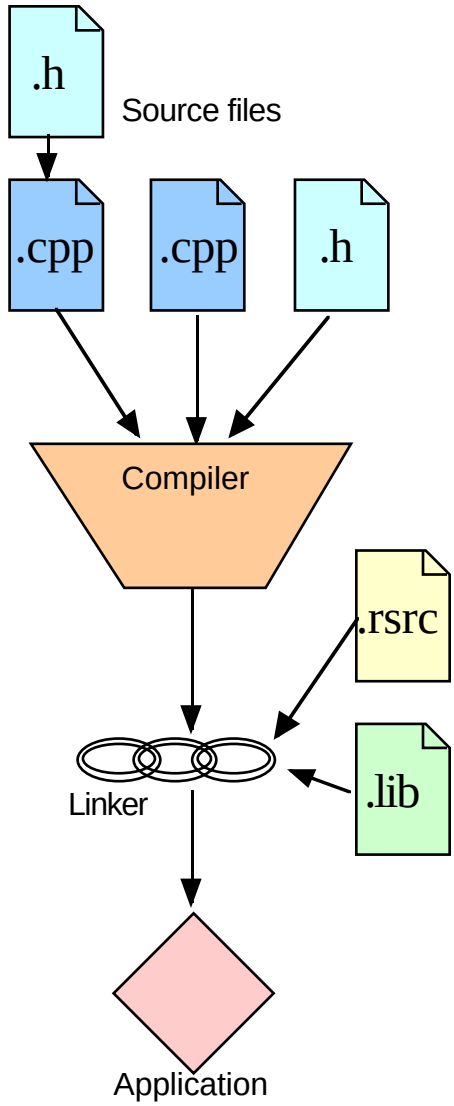
This should help you to have the confidence to be able to move on, and get through this chapter!

Project Files

Project Files are the key to programming any new application. This file is really just a list of all of the components that are used to build the application. Note- it's just the LIST of components, not the components themselves. The components that actually make the application are held in other files, of various kinds. These components may contain:

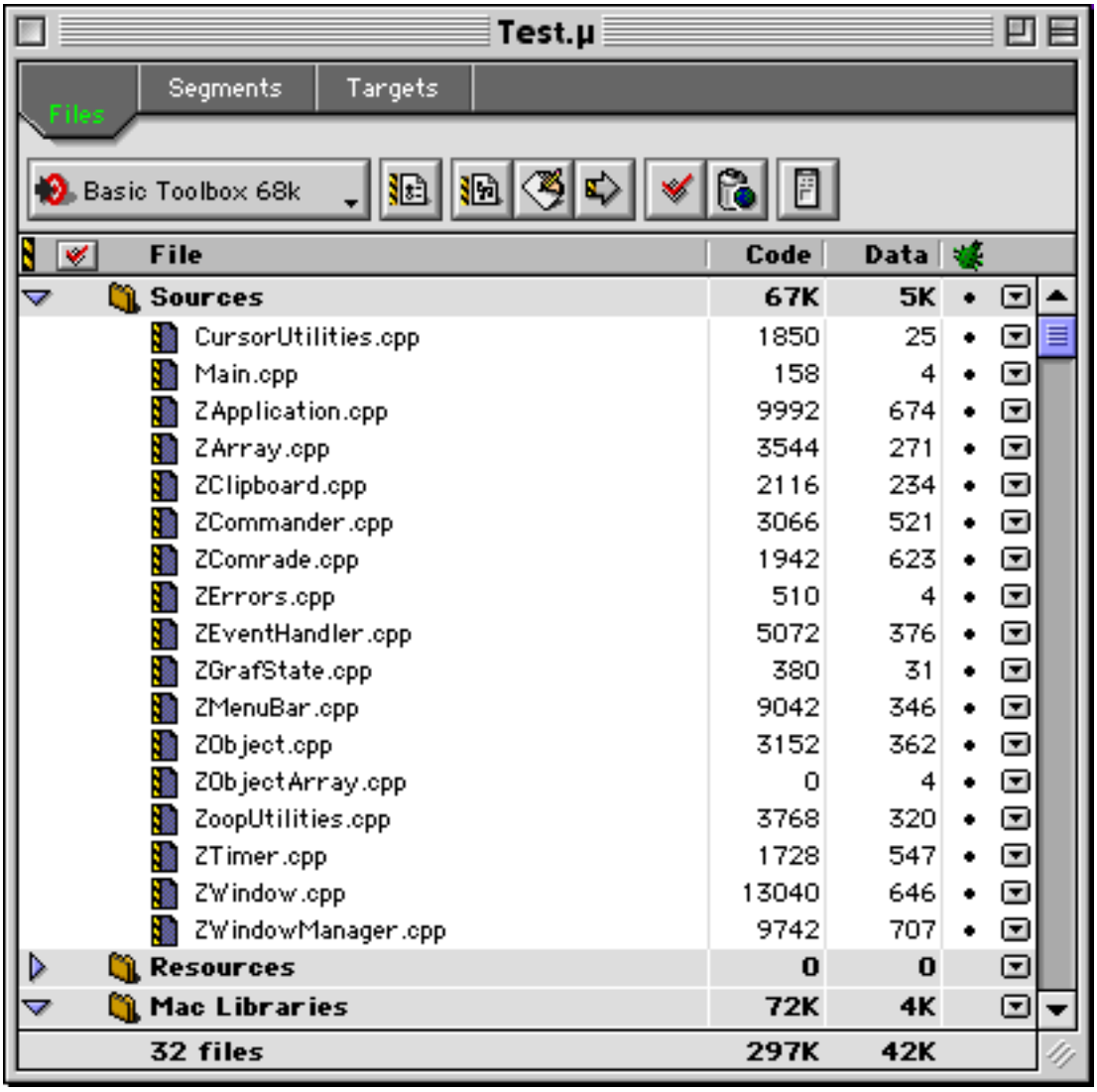
- Source Code
- Compiled code (object code)
- Library code and stubs
- Resources

The job of CodeWarrior is to take all of these disparate components, and turn them into an executable application. (We could also create other types of executable file, but we'll disregard that for now- MacZoop is used to make applications). The main job involved in doing that is to turn the source code into executable (or object) code. Note that the word "object" here has nothing to do with object-oriented programming. To avoid any confusion, we'll call it executable code from now on. Here's a diagram to show what CodeWarrior does.



Now, how does CodeWarrior know which particular files out of the thousands on your hard disk are required to build a particular application? The answer is, the Project File tells it. As I said above, the Project File is really just a list of other files. Some development systems use a file called a “Make File” to tell the system what components are to be built. Apple’s MPW is such a system, and it is even possible to use MPW to compile MacZoop, but that’s not really recommended for beginners, so we’ll come back to that later. Traditional Make files are usually just text files that are edited as any other, and have a particular syntax that the development system understands. The trouble with Make files is that they are a bit messy and error-prone, and so Metrowerks decided to do away with them for CodeWarrior, and allow the list to be set up much more easily by simply adding files to a list in a window. This list is saved in a private format, and that’s what the Project file is. It’s a pity that the format wasn’t more carefully managed as CodeWarrior evolved, but we can’t do much about that now.

So, let’s look at what a basic MacZoop project looks like, when opened in CodeWarrior. Here’s what it looks like in CodeWarrior Pro 1- yours might be a bit different, but not too much.



Now, while MacZoop comes with project files for the basic and demo projects, they may not be any good to you, because of the format problem. You may find that your version of CodeWarrior is too old or too new to open them. Since we are resigned to this, we may as well suppose that they are not going to work, and we'll have to set up the projects from scratch. Once you've done this, you can re-use the project files created by your particular version of CodeWarrior to start new projects any time you need to, so the stuff we're going to do now will probably only need to be done once.

Creating the Project file

Again, different versions of CodeWarrior do this in different ways. If this doesn't make sense, read the manual for your particular version. There is always some way to do it!

Create a new project file using the "New Project..." menu command, or maybe you choose "New..." and there's a button to select new project or new file. It is rare to create a project completely from scratch, so usually you base it on some stationery that is already preset with some of

the basic things you need. Again, this varies significantly with the version you have, so consult your manual. Later versions are better in this respect than the older ones. In Pro 1 for example, a good stationery choice is “Basic Toolbox PPC” or “Basic Toolbox 68k”, which is found in the C/C++ section under MacOS in the stationery choice window that comes up. You could also choose “Basic Toolbox Multi-target” if you’re creating an application for both PowerPC Macs and 68k Macs, but this is slightly more advanced, so for now pick the type of Mac you have. If you don’t know, choose the 68k version. If you can’t see anything like this, choose something that seems appropriate, such as ANSI C++, etc. Then you’ll be asked to name the project file and save it on the disk. At this point, so that you can follow the tutorial without any problem, it’s important to put the file in a sensible place. The sensible place is within a new folder inside the “Projects” folder in the MacZoop hierarchy. So, to do this, navigate to the Projects folder (MacZoop 2.1 *ƒ* -> Projects). Hit the “New Folder” button, and give your project folder a name- anything you like. You should now be inside that new folder, e.g. MacZoop 2.1 *ƒ* -> Projects -> My New Project. Give the project file itself a name. Now, by convention, we programmers usually give project file names a particular form. This is just to aide us when dealing with the files later on. It’s a good idea to form this habit, so let’s do that- we give the file a name followed by .µ. That’s the greek letter “mu”, and it is obtained by typing the ‘m’ key while holding down the option key. So our file will be called something like “MyProject.µ”.

Once done, the new project window will open. Later versions of CodeWarrior can open more than one project at a time, earlier ones will only let you open one at a time. The project so created has a few components listed already, and may be organised into groups, such as Libraries, Resources, etc. It is likely to contain some source code files called “SillyBalls.c” and a resource file called “SillyBalls.rsrc”. These are not useful to us, and in a minute they will be thrown away- they are just placeholders for the useful files you want to add. Again, your mileage may vary- some versions of CW have files called <ReplaceMe.c> and so on.

Adding source files

We need to add the basic source files that every MacZoop project requires. These are all contained in the “Required Classes” folder within the MacZoop 2.1 *ƒ* folder. There are two ways to do this, using a dialog resembling the File Open dialog, or by dragging and dropping. Drag and Drop is definitely more convenient, if your version of CodeWarrior supports it. Let’s try- open the “Required Classes” folder in the Finder, making sure that you can see the Project window behind. There are quite a lot of files in the Required Classes folder. They are all text files, but there are two distinct kinds. There are header files, which all end in .h, and there are source files, which all end in .cpp. For convenience, each type has had a different label assigned to it in the Finder, so they can be sorted into two groups- headers and sources. So however you prefer to view the files, use the Finder to sort them by Label, to make this next bit easier. While header files are vitally important, we don’t need them right now. What we need to do is to pick up all of the source files, and add them to the project window. We do this by selecting them all, then dragging them over the Project window in CodeWarrior. This should highlight, and in addition, show us where in the window they will be inserted. We want to add them to the same place we see the SillyBalls or ReplaceMe file. Make sure that’s highlighted, then drop the files.

If you haven’t got drag and drop support in your version, you need to do it this way instead. Choose the Add Files... command from the Project menu in CodeWarrior. A dialog will appear which will allow you to navigate to the “Required Classes” folder. Add all of the .cpp files by selecting each one and clicking Add. Once all of the .cpp files are added, click Done and the files will appear in the Project window. They may be in the wrong place, in which case select them

again within the project window, and drag them to the right place.

Either way, you end up with the same thing- the project window now lists all of the .cpp files from the “Required Classes” folder. Next step is to delete the SillyBalls (or ReplaceMe) files. Highlight them, then choose Remove Selected Items from the Project menu.

These are the files you need to have added:

CursorUtilities.cpp
Main.cpp
ZApplication.cpp
ZArray.cpp
ZClipboard.cpp
ZCommander.cpp
ZComrade.cpp
ZErrors.cpp
ZEventHandler.cpp
ZGrafState.cpp
ZMenuBar.cpp
ZObject.cpp
ZObjectArray.cpp
ZoopUtilities.cpp
ZTimer.cpp
ZWindow.cpp
ZWindowManager.cpp

This list is correct for MacZoop 2.1. If you are using this manual in conjunction with a later version, there might possibly be one or two extra files, but as long as you add all the .cpp files in the “Required Classes” folder, it should work OK.

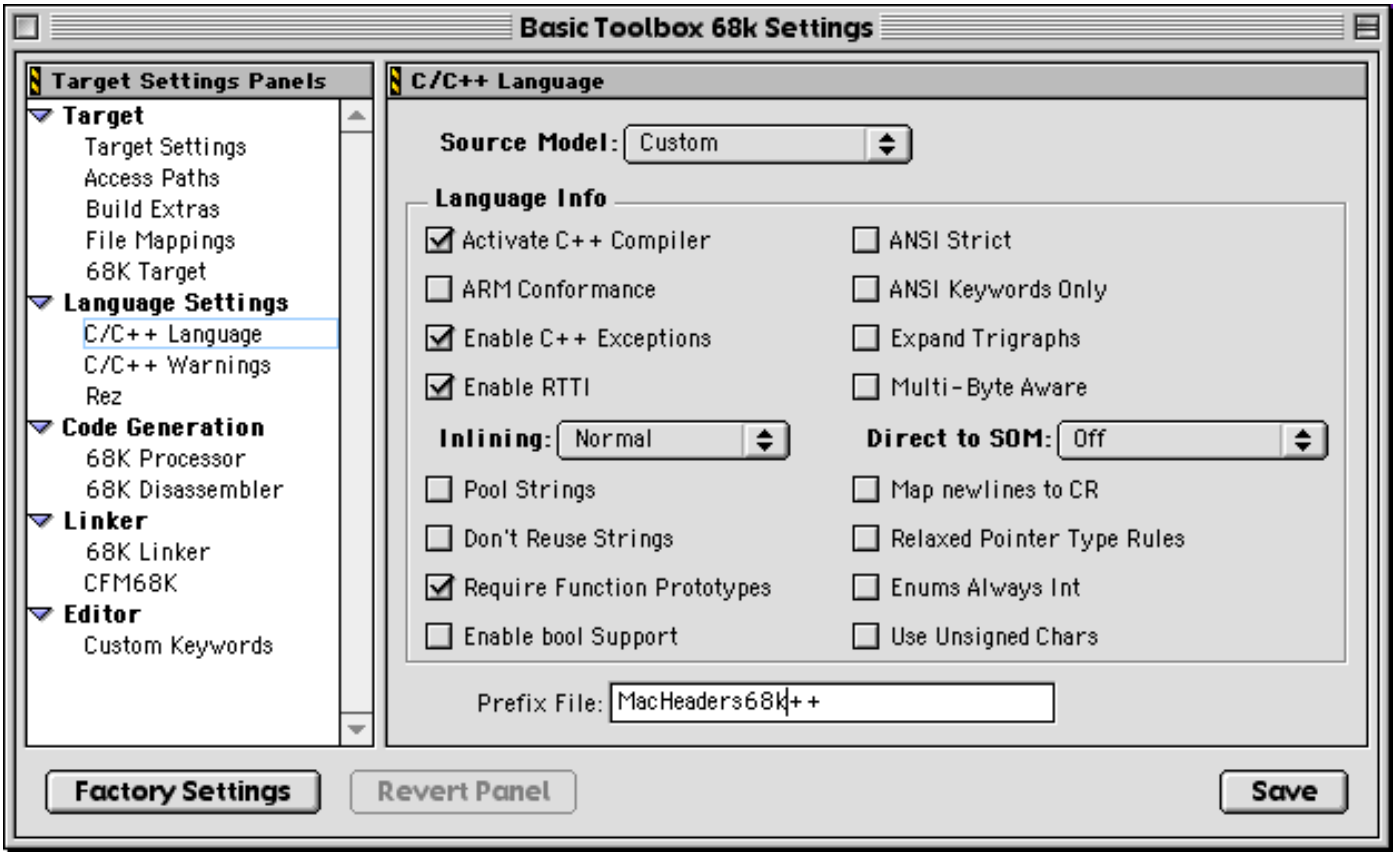
We need to repeat the exercise now for the resource file that MacZoop needs. There is only one required file, and that is located in the Projects folder. It is called “MacZoop.π.rsrc”. Either drag and drop it on the Project window, or else use Add Files... to add it, as before. Now get rid of the SillyBalls.rsrc file, or the ReplaceMe.rsrc file.

CodeWarrior Project Settings

The project file in CodeWarrior contains more than just a list of files. It also contains the compiler settings needed to build what we want. We need to make sure that these settings are correct, since if not, we will have problems compiling the MacZoop source code. Once again, differences between CodeWarrior versions cause us a little grief here, since access to these settings has changed from version to version. However, you’ll find them either in the Preferences dialog, or else a dedicated “Settings” dialog. For example, because I created my project using stationery in CodeWarrior Pro 1, my settings show up under a menu item “Basic Toolbox 68k settings...” under the Edit menu. You should find something similar. If totally stuck, you’ll have to look at the manual for your version of CodeWarrior.

The settings are divided up into a number of groups in the dialog. The most important of these is under the “C/C++ Language” panel. Here, we are setting the very basic settings that the compiler

uses when trying to make sense of our source code. Here's what mine looks like, after the correct settings have been selected:



The important checkboxes are:

• Activate C++ Compiler must be ON (naturally since MacZoop is written in C++)
• Enable C++ Exceptions must be ON (since we use exceptions in MacZoop)
• Enable RTTI must be ON (since MacZoop uses RTTI)

The other important thing here is at the bottom- the "Prefix File" setting. This should be:

MacHeaders68k++ for 68k projects
MacHeadersPPC++ for Power PC projects

At least this is true for CodeWarrior Professional (all versions as far as I know). Older versions may have called this file something different. In the dim past it used to be just called "MacHeaders". If in doubt, please consult your CodeWarrior documentation. This does need to be right! After checking these settings, click Save.

The next important panel is the "68k (or PPC) Target" panel. This is where the basic application parameters are entered. CodeWarrior uses this to preset your application with some important information when it makes it on the disk. The project type should naturally be set to "Application". If not, you must have used the wrong sort of stationery. If this is the case, you'll probably have to go back and start over, since very little will be set up correctly, and putting it all right will be a bigger job than starting from the beginning.

Set the other settings thus:

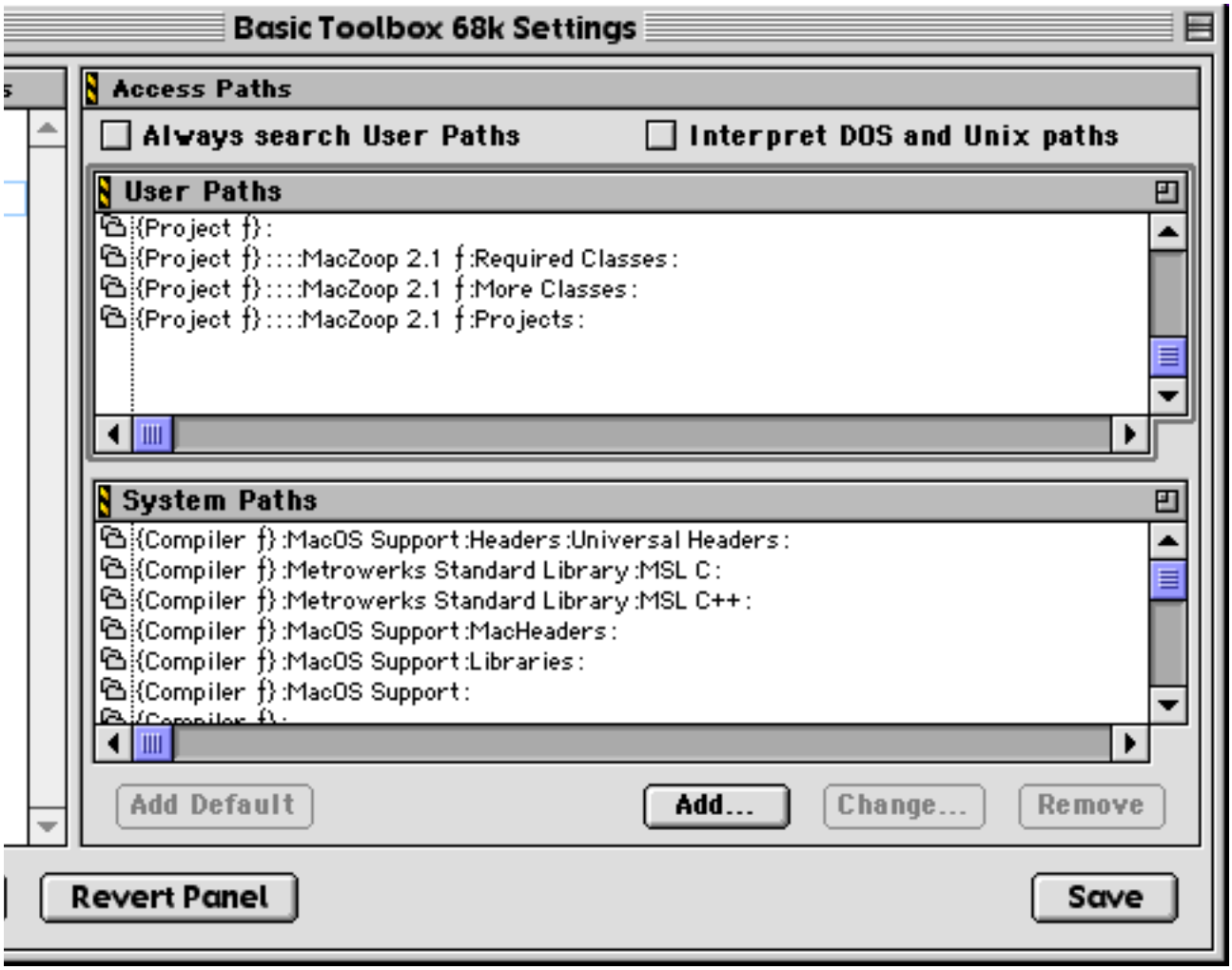
Creator:	ZAPP
Type:	APPL
Preferred Size:	1024K
Minimum Size:	384K
Startup Code:	Standard

In the SIZE flags menu, the following should be checked:

- acceptSuspendResumeEvents
- canBackground
- doesActivateOnFGSwitch
- is32bitCompatible
- isHighLevelEventAware
- localAndRemoteHLEvents
- isStationeryAware

All others can be left unchecked. Later, when you build your own apps, you may want to set other flags here, but for now, we don't need anything too special.

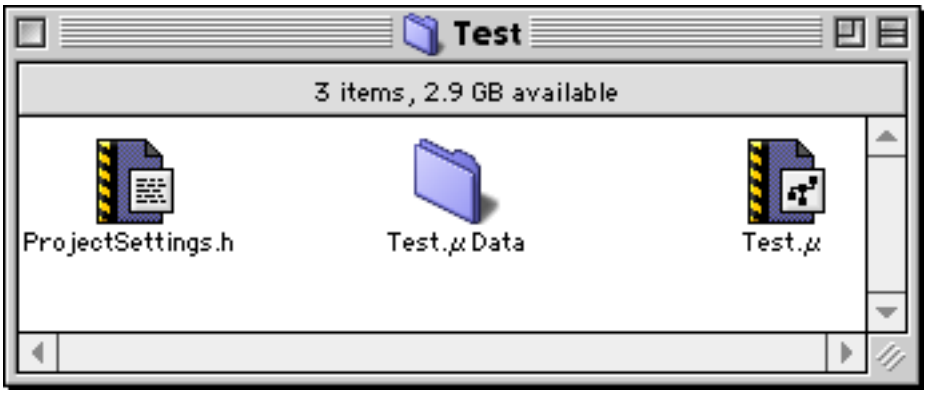
Set the Filename field to what you want your application to be called- e.g. "MyFirstApp". After all this, click Save.



Next settings to check are the Access Paths. This panel allows you to tell CodeWarrior where to look for files while building the application. There are two sections, project paths and system paths. We don't need to change the system paths, and the Project Paths are already set up with one basic path. We do need to add a couple of others though.

Click in the project paths section to make sure it's selected, then click Add... Navigate to the "Required Classes" folder. Make sure "Project Relative" is set and click OK. Repeat, but this time navigate to the "More Classes" folder. Finally, do the same for the "Projects" folder itself. Your window should look something like the screen shot above.

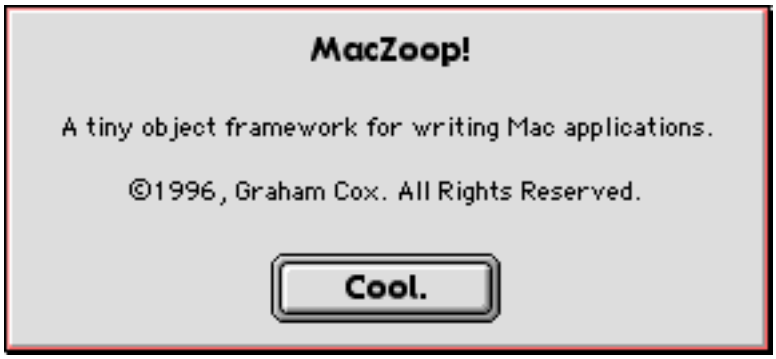
OK, we're almost ready to go, but there's one more thing we need to do. Every MacZoop project has its own settings that allow you to customise certain features. This is used to tailor the general purpose MacZoop code to your own needs. For now, we don't need to change these settings, but we do need to make sure MacZoop can read them. The settings are held in a file called "ProjectSettings.h", and each project has its own individual copy of this file. To arrange this, we go to the Finder, and open the Projects -> Basic MacZoop Project folder. Find the ProjectSettings.h file, and COPY it to your new project folder (to make a copy, hold down option as you drag the file). Now your project folder will have a number of items, like this:



Don't worry if it looks a bit different- again this is probably due to differences with CodeWarrior.

We're Ready to Roll!

OK, all we need to do now is compile the application! Choose "Make" from the Project menu, and CodeWarrior whirrs into action, compiling each file, linking together all the code, and writing the application to the disk. If you got problems, we'll sort you out in a minute, but with care, and a little bit of luck, you won't see any errors, and everything is fine. All we need to do now is to run the application! We could find it on the disk and double-click it, but CodeWarrior helps us by running it for us. Choose Run from the Project menu, and your MacZoop application will launch. What you should see is a blank, empty window, and the usual Apple, File, Edit (and Help) menus. If you look under the Apple menu, there should be an "About MacZoop..." command- choose it, and you should see:



Congratulations! You're in business! That wasn't so bad now was it? Use the New command to open lots of windows if you want... that's about all it does mind!

Troubleshooting the Application.

Even if compilation went perfectly, the app itself may not work. There are usually only a few things that can be wrong, since most errors are picked up at the compilation stage.

• The menus appeared, but the window didn't. Go back and check your 'SIZE' flags in the Target Settings panel. The 'localAndRemoteHLEvents' needs to be ON in order for the Finder to send the application the right message to tell it to open the window.

• The "Run" command produced a brief flicker, but nothing seemed to launch, and there wasn't an error. This is most likely because you didn't include the "MacZoop.π.rsrc" file- the application has been created with no resources! In this situation, the application hasn't even got any way to tell you what the problem is, because there isn't even the most basic error alert resource available. Check that you have included the resource file and try again.

Troubleshooting Compilation

If things didn't work out, it's more likely that the problem showed up while compiling the application rather than running it. There are two basic types of error:

- Compiler errors
- Linker errors

You may have got only one type or the other, sometimes both. Make sure you know what it is you're trying to fix!

Bad Settings

Compiler errors are due to a problem interpreting the source code itself- something doesn't make sense. Linker errors are due to a problem connecting up the various bits of code into a coherent whole. Sometimes you'll see hundreds of errors, and it looks like a daunting task to find and fix them all. Don't worry- there are very unlikely to be hundreds of errors. In fact, I know that when MacZoop was released, it compiled just fine, so it's extremely unlikely that hundreds of errors have actually crept in there. The fact is that C and C++ compilers are actually pretty stupid, and it sometimes takes just one little thing to throw them- they plough on through the code regardless

and generate errors because they are not interpreting things properly. So, if this is happening, first thing to check is the language settings. See above. It's not likely to be able to compile C++ if the language is set to only C. Likewise, MacZoop uses C++ features such as Run Time Type Identification (RTTI) and Exceptions that have to be turned on, otherwise source code that uses those things will not make sense to the compiler.

Missing Headers

The next most common cause of compiler errors are missing header files. Header files are source code files that define all sorts of useful things, such as data types, function prototypes, class definitions, and so on. They are vital! Header files for MacZoop come with MacZoop, naturally enough. However, there is another set of header files needed by MacZoop, and they come from Apple. Those headers define all of the toolbox types, functions, etc, so that the compiler knows what we're talking about when we talk about Rects, for example, or Menus. These header files come with CodeWarrior, and over time, as you might expect, have changed as new things were added to the toolbox. MacZoop can use various versions of these headers, but very old versions of CodeWarrior may come with such old files that a problem arises. This is not a problem however, since Apple makes the latest headers available for free to anyone who wants them, and any version of CodeWarrior can use the latest headers. Thus is is prudent to use up-to-date headers (known as the Universal Headers), even if you are working with a very old version of CodeWarrior.

Now, updating the headers is not particularly difficult, but again, varies with CodeWarrior version. If you think you need to do this, then please refer to your CodeWarrior documentation. If you download the headers from Apple's main developer website, there are also some instructions there that tell you what to do. One step of the process is to recompile MacHeaders based on the new headers- your version of CodeWarrior will come with projects and instructions allowing you to do this.

This is one of the most common causes I've seen of the "Help! I've got three thousand errors!" variety. A few years ago, Apple spent quite a lot of time modernising the headers, and many changes were made. The transition was a bit painful, but worth the effort, since they are much more logically organised these days. However, the after-shocks of those changes are still being felt occasionally, and this is likely to be one of them. Bite the bullet- get the newest headers, rebuild MacHeaders, and carry on. Once done, you probably won't need to do it again.

Bad Paths

The final reason that you may get these errors is that the headers are all OK, but CodeWarrior can't find them. This is what the "Access Paths" set up we did was for- it was telling it where to look. The System Paths already have paths to Apple's headers, but we had to add those for MacZoop. Check that you put these in correctly. The headers that MacZoop absolutely has to see are in the Required Classes folder, with the exception of ProjectSettings.h, which is individual to your project.

Linker Errors

This class of error has a different cause. Very often, we call upon code external to our application to perform some function. The toolbox itself is the prime example, but there are others, such as libraries, shared libraries, plug-ins, etc. As with headers, we have to tell the compiler where to

look for this code, and failure to do this is the usual cause of this error (which almost invariably come in packs, like wolves!). For example, if we want to use QuickTime in our application, we link to the QuickTimeLib library. The compiler is happy to accept a call to QuickTime when it encounters it- it knows that it will probably be supplied later on, so no error is generated at that time. Only later, when the Linker tries to actually connect our code to QuickTime, it finds the library isn't there. This doesn't mean that you don't have QuickTime installed on your machine, it just means you forgot to tell CodeWarrior about it. Usually, small files called "stub libraries" are used to stand for the real library when we compile our applications. These stub libraries are supplied with CodeWarrior for all of the current Mac functionality that exists. However, new things come along, and to use them, it may be necessary to use a new stub library for the new functionality. These will be provided by Apple (or third parties) with the SDK for that particular piece of functionality. The stub libraries are added to the project like any other source file.

Solving Linker errors can need a bit of detective work, because the basic problem is that some information is missing. Therefore the error message can't always tell you what you need to do to put it right, only what's wrong.

The Next Step

While the basic project doesn't do much, it's an important step to be able to compile and run it properly. Once you've got this far, you're pretty much in business. It's worth saving the created project file as stationery- you can then use this to create other MacZoop projects in the future without much effort- you know you are starting with a working basis. That's important for productivity, and confidence in the framework and your tools.

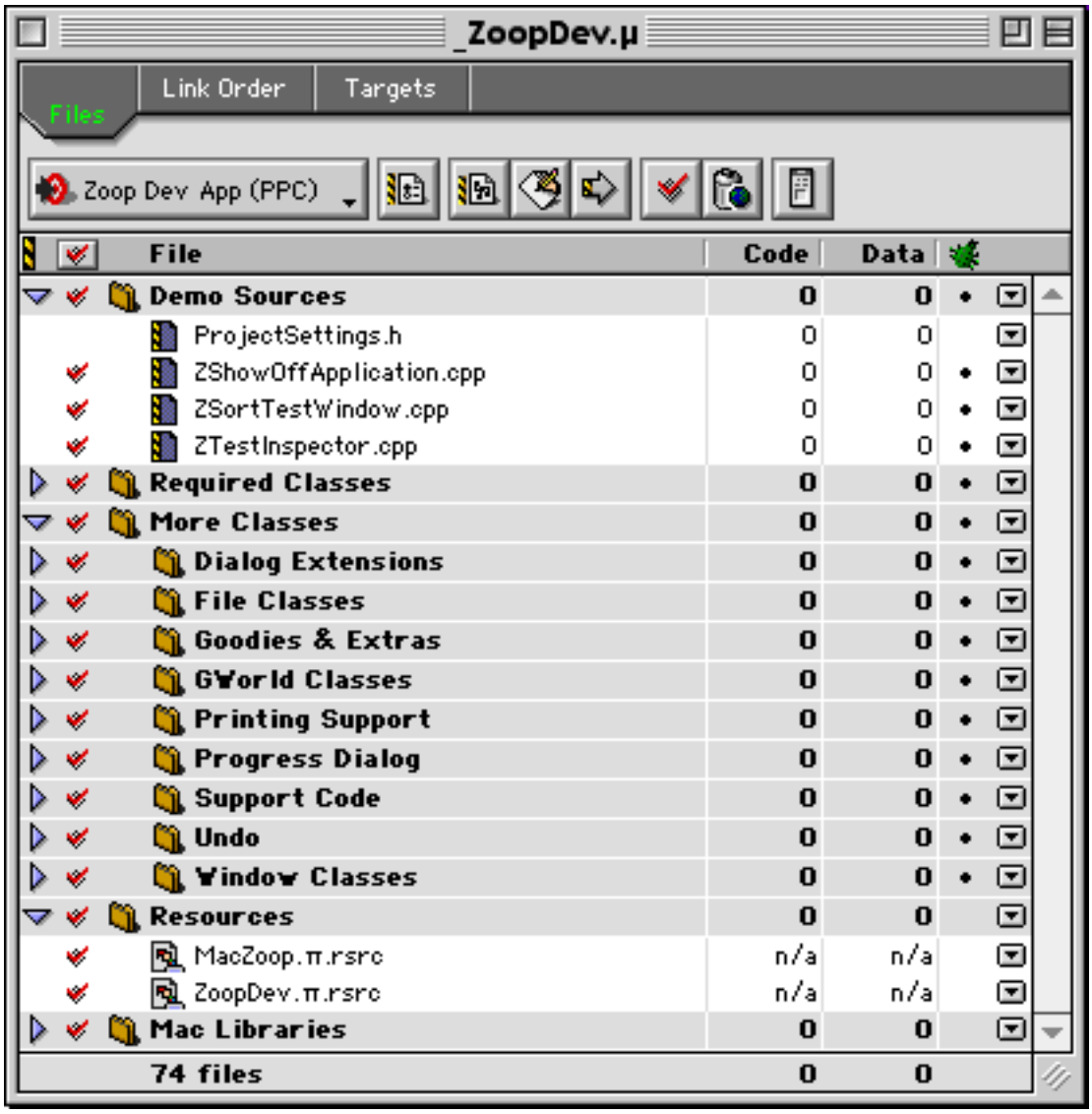
You are now ready to build a more sophisticated project. You can go on to try your hand building the demo project, or do the "Hello World" tutorial (next chapter) which is where you will start to learn how to use MacZoop itself.

Demo project

The demo project serves a number of purposes. First, it is a place where new developments in MacZoop can be tried out and tested. This is important for me to ensure that MacZoop is of consistently high quality and contains as few bugs as possible. Second, it allows those various features to be demonstrated. However, sometimes a particular feature can look rather unimpressive until you come to use it in anger yourself. Finally, it provides a project that is relatively complex- if you can compile and run it successfully, you really are set to get on and use MacZoop in earnest. So let's see how to do that, and along the way, we'll look at a few of the simple ways you can customise the application you build. None of this requires that you write any code, or understand how MacZoop works.

The MacZoop distribution comes with the demo project complete, but as before, you may have to recreate the project file if your version of CodeWarrior is not able to open it. If this is the case, make a copy of your basic project file and move it to the Demo Project folder, and rename it suitably. Open this file in CodeWarrior, and edit the application target name as required. You now need to bring in the other files needed for the demo.

This screen shot shows what the demo project window looks like as supplied.



This project includes many more source file than the basic project. Most of these are found within the “More Classes” folder, which is itself divided into groups representing certain types of related functionality. You’ll need to locate all of the .cpp files and .rsrc files in this hierarchy and add them to the project. Remember, drag and drop them, or use the Add Files... command. You also need to add the classes and source code that is specific to the demo. These files are only used in the demo- they are special versions of ZApplication called ZShowOffApplication, and a couple of demo window classes ZSortTestWindow and ZTestInspector.

The demo is usually set up to use a number of extra libraries that the basic project doesn’t use. These are (for PPC):

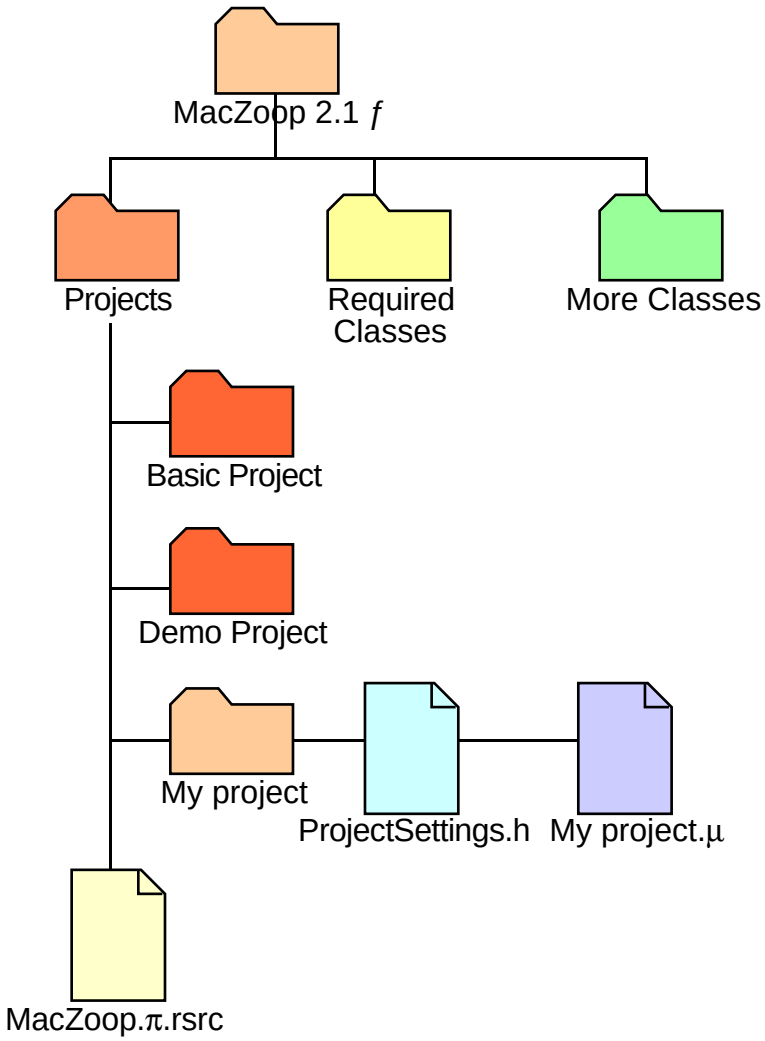
- AppearanceLib
- ControlsLib
- DragLib
- QuickTimeLib

- StdCLib
- NavigationLib

This should compile and run without any problem. If you do experience difficulties, use the troubleshooting information above to see if you can spot the problem. If you get linker errors for one of the more esoteric libraries, you may be able to press on simply by avoiding those features. For example, MacZoop can use the new Navigation Services for picking files, but it doesn't have to- you can turn off this feature and use the Standard File package instead. This is done using the ProjectSettings.h file. This contains numerous switches and values that affect the way the application is compiled. One of these is _USE_NAVIGATION_SERVICES. You can set this to OFF, and you can avoid any linker errors to do with the NavigationLib. Similarly, there are switches like this for Appearance and other relatively modern Mac features. ProjectSettings.h is examined in detail in a later chapter.

It is well worth trying to get the demo to compile and run successfully. Once you have the basic project working, this should be fairly painless, but since the demo represents a much more complete and realistic application than the basic project, should help give you confidence before you tackle your own application development. There are no real pitfalls beyond those already discussed, and errors should be tackled in the same way.

Project Organisation



The key to success with your own application projects is to be reasonably organised. While MacZoop doesn't impose much order of its own on you, organising things sensibly will more closely match my expectations when I wrote it- that way, the opportunity for unexpected errors to occur is much reduced- after all, I can't test for every possible way a person might organise their work.

The Projects folder is where you should put your projects. By making folders as needed in here, and keeping project-specific code within your own folders, you can extend the MacZoop folder hierarchy sensible. The advantage is that all projects require the same access paths, so if in doubt, you can refer to an already known to work project. Here's the expected (and suggested) layout.

Every project should have its own ProjectSettings.h file. This should reside at the same level as the Project file itself.

No project should require special changes to the basic MacZoop source code. The great advantage of a framework is that the basic code remains untouched and common to everything. If you ever find it necessary to alter a MacZoop file, make a copy of it in your local project folder and only edit that copy. That way your other projects are protected from the unforseen consequences of a change.

Every project includes the MacZoop.π.rsrc file, which contains the basic resources that MacZoop always requires. In addition, your own projects will probably add to the resources- if so, create these in a separate file, and add it to your project. It's possible (and expected) that you will want to substitute your own resources for some of the supplied ones. Rather than edit the common resource file, simply put the resources you need in your own file, then make sure that these resources are given priority over the supplied ones. How? Well, the Link Order tab in the project window lets you set the order that files are linked in. By making sure that your resource file is further up the list than the supplied file, your resources will override any duplicates in the basic file. Earlier versions of CodeWarrior didn't have a separate Link Order list- in this case, simply make sure that your file is above the basic file in the main list.

The (in)famous "Hello World" tutorial!

In this chapter, we are going to see how we use MacZoop to actually do some real programming. Before you start this tutorial, make sure that you can compile and run the basic project without any problem. If there's something wrong with your set-up, you need to fix it before attempting this- that way you won't be trying to hit a moving target. You may also like to try to compile and run the demo project. This isn't required for the tutorial, but it may help give you confidence in your set-up and MacZoop in general. It's up to you...

OK, this tutorial guides you through some basic principles of MacZoop, and object-oriented programming in general. We are going to use MacZoop to create a custom, standalone Macintosh application, which will display the immortal words "Hello World!" in large, friendly letters in a nice on-screen window. While not of much intrinsic value, this will teach you:

• How to customise MacZoop to create your own window types rather than generic ones
• How to subclass a standard MacZoop object
• How to override object methods to do useful work
• A little about how MacZoop applications are put together

A brand new project....

Create a new project by cloning the basic MacZoop project. Remember, create a new folder in your projects folder (call it "Hello World project" for example). Make sure you have a project file and your own ProjectSettings.h file. If you cloned the basic project folder, discard all the other stuff in it for now.

Open the new project file in CodeWarrior. Open the settings, and change the application name (in the Target panel) to something suitable- say "Hello World". Now, before doing anything else, make sure it compiles and runs cleanly without any problems. If this is not the case FIX it first! (refer to the troubleshooting information in the previous chapter).

The Window class

The first thing we need to do is to define our window class. This is the object that will display the "Hello World!" text to the outside world. We don't need anything too fancy- just a basic window with enough space to show the text. In fact the built-in window template will serve just fine. However, the built-in window object- ZWindow- doesn't display anything except a vast expanse of crisp white, so we need to make a new class that will do this job. Since ZWindow already knows a great deal about how to be a window, we will use this as a starting point.

So:

• Open a new blank text file. This will become our class header. To do this, select New from the File menu in CodeWarrior.

Now, it's not actually necessary to do this, but it is a good habit to form- add a comment to the top of the file that describes what it is. All MacZoop header files have a comment like this, so a good shortcut is simply to open one of them up and copy and paste that part into the new file. Then edit the info as you require.

This is going to be a header file, and we need to ensure that header files are only actually read by the compiler once. Not only is this more efficient, but many compilers will get confused if they read the same header more than once. So, below the comment area, add the following:

```
#pragma once

#ifndef __ZHELLOWINDOW__
#define __ZHELLOWINDOW__



#endif
```

Eh? Right, what we've done is provide two ways to make sure the header is only read once. The line #pragma once tells the compiler- read this only once! For CodeWarrior, that is enough actually. But, MacZoop is not required to use CodeWarrior, and other compilers (particularly older ones) may not understand this instruction. So, just to make sure, we also add a more traditional way to ensure that the file is parsed only once (parsed is a computer nerd's way of saying 'read'). The other lines say IF you've never seen the word "__ZHELLOWINDOW__" before, then here it is... if the file has been read before, then the computer will have seen it before, and will then skip to the #endif, skipping anything in between. It's in between that we are going to put our interesting stuff. By the way, that's two underscore characters on each end of ZHELLOWINDOW. Actually some compilers treat words defined like this with double underscores in the same way as the #pragma once, so we've actually covered ourselves three ways here- all good practice, and I recommend it's a habit you get into early.

OK, now for the more interesting stuff. Remember, everything we add must go after the #define __ZHELLOWINDOW__, and before the final #endif.

Type:

```
#include   "ZWindow.h"

class ZHelloWindow    : public ZWindow
{
public:
      ZHelloWindow( ZCommander* aBoss, const short id );

      virtual void    DrawContent();
};
```

Whoah! OK, let's look at this line by line.

`#include ZWindow.h.`

This makes this header bring in another one- the one for ZWindow. That header is needed because it defines ZWindow, and as we are basing our new window on that one, so we need its definition.

`class ZHelloWindow : public ZWindow`

This tells the compiler we're defining a new class called ZHelloWindow, and we want it to be based on the public interface of ZWindow.

```
{
public:
```

This tells the compiler that the methods and data members we define following this word are part of our own public interface. Actually this public and private stuff is not really very important right now, so don't worry about it for the moment.

```
ZHelloWindow( ZCommander* aBoss, const short id );
```

This looks familar- it's just a function prototype. However, because the function has the same name as the class, it is treated as a special function. In fact this is the CONSTRUCTOR of the class- it is the function that will be called automatically when the class is created. We'll see how this is useful later. Functions that are defined within a class like this are called the methods of the class. Here, we can see that this method takes two parameters- a ZCommander*, which is another MacZoop type, and a short. We don't need to worry about this for now, since for this tutorial, all of that is taken care of for you.

```
virtual void DrawContent();
```

Ah, this is more interesting. This is another method of our class. It takes no parameters, but it is declared as a VIRTUAL method. This means that it can be overridden by a subclass if desired, but more importantly for us here, it means that we are overriding the similar method of ZWindow. This is where we are going to do our special drawing!

OK, we're done here. Save the file to your project folder, and be careful to name it "ZHelloWindow.h". Don't be tempted to give it another name for fun- the naming is important as we'll see.

So we have our new class defined, but we still haven't written any code. Let's do that now.

• Open another new blank file.
• Cut and Paste a comment block to describe the file. Edit the comment as you wish.
• Type:

```
#include   "ZHelloWindow.h"
#include   "MacZoop.h"

ZHelloWindow::ZHelloWindow( ZCommander* aBoss, const short id )
     : ZWindow( aBoss, id )
{
}
```

The first two lines simply include a couple of header files- the one we just created, and the MacZoop.h header file, which defines all sorts of useful MacZoop stuff that we might want to use. In fact we probably don't need it in the tutorial, but again, this is a good habit to get into since most real-world files will need it.

The next bit is the body of the CONSTRUCTOR. In fact, we don't need it to do anything special, but it must call the constructor of ZWindow, which we are built on top of. That's what's happening here- ZWindow's constructor is called, and the <aBoss> and <id> parameters are simply passed on to it. Now the interesting bit.

```
void ZHelloWindow::DrawContent()
{
     TextFont( kFontIDTimes );
     TextSize( 24 );
     TextFace( bold + italic );

     MoveTo( 10, 50 );
     DrawString( "\pHello World!" );
}
```

This is the body of the DrawContent method for our class. It sets up the text drawing parameters, moves the pen location to 10 pixels horizontally and 50 pixels vertically, then draws the text string “Hello World!”. That’s all the code we need to write. Because we are based on ZWindow, we don’t need to worry about how to make the window appear, or how it should behave- all we need to do is concentrate on how our particular window is different- ours draws “Hello World!”, and that’s all we’ve coded. Right, save the file, as before, to your project folder, and name it “ZHelloWindow.cpp”. Again, be sure to name it correctly.

Now we need to add our new source file to the project.

• With the new file frontmost, choose “Add Window” from the Project menu. You should see it appear in the project window. You can drag it wherever you want to suit yourself.

If you now compile and run your project, you should find that your code is compiled OK. If not, now is the time to go over it again and FIX it. Is that all that is needed? No. When you run your application, you get a blank window. That’s because MacZoop is told to make ZWindow objects as standard, and you haven’t told it anything about ZHelloWindows yet. So, quit the running app and follow on...

Project Settings

Your ProjectSettings.h file is the place you tell MacZoop what sort of window to make. Since this isn’t part of the project, you have to open it by either double-clicking it in the Finder, or using the Open File... command in the File menu of CodeWarrior. As a shortcut, you can add this file to your project. Since it’s a header, it can sit in the project file quite happily and not really have any effect, but it gives you a convenient way to open the file whenever you need to.

ProjectSettings.h contains a good number of tweakable parameters which affect how your app is built. What we are interested in here though is the window type to make. By default, no window type is specified. MacZoop knows to make generic ones in this case. Scroll down to about line 50. Actually, it doesn’t matter where you add this, but the supplied file already has what you need on line 50, but commented out. It reads:

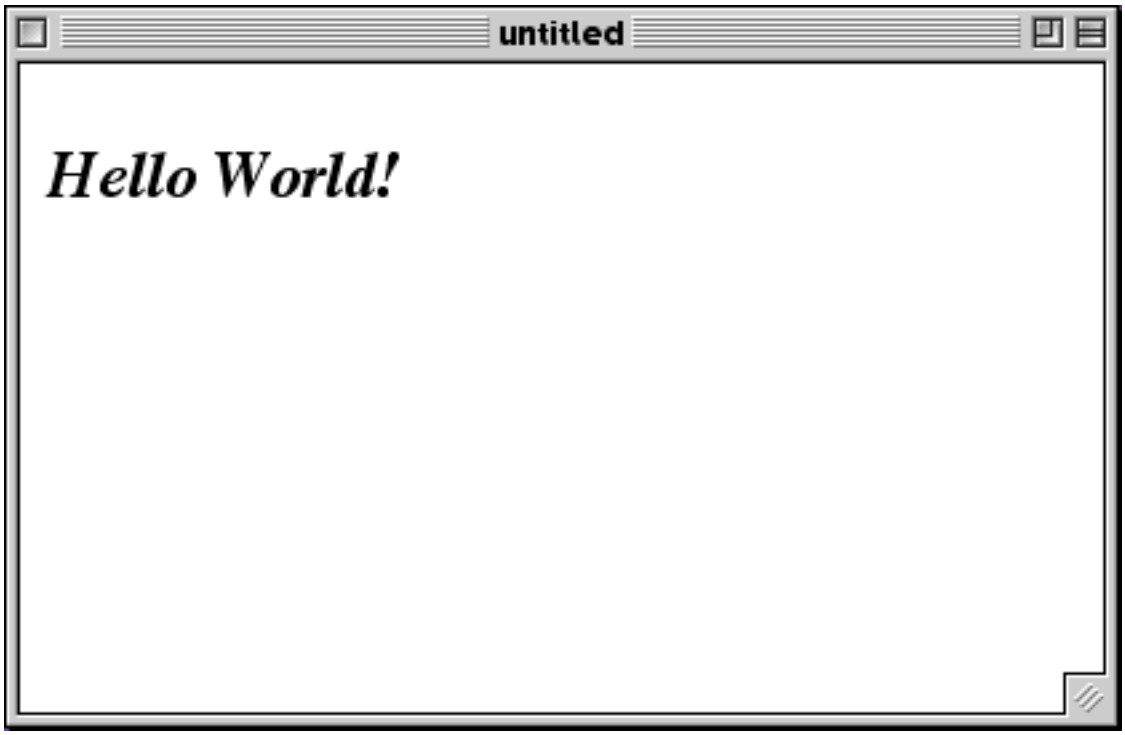
```
// #define      USER_DEFAULT_WINDOW_TYPE   ZMyFunkyWindow
```

change this to read:

```
#define         USER_DEFAULT_WINDOW_TYPE   ZHelloWindow
```

Note- we took away the comment slashes, and changed ZMyFunkyWindow to ZHelloWindow. When CodeWarrior encounters this line, it causes it to compile the code for ZApplication a little differently. It automatically #includes the file ZHelloWindow.h, and when instructed to do so, it

makes a ZHelloWindow object instead of a ZWindow object. This is why the naming of the files and the class was so important- by putting that name here, MacZoop can find all the right files and make the right object. Save and close the file, then compile and run it again. This time, you should see:



Congratulations! You have successfully completed the Hello World tutorial! Move the window partly offscreen, then on again- the text is refreshed. Open more windows using the New command- each one is a ZHelloWindow- we didn't just tell it how to create one window, but as many as we want of that type. As the various windows are moved around, you'll see that they refresh properly, grow shrink, close and generally behave as windows should. Yet the only code we wrote was to write the text! This will, I hope, give you a glimpse of the power of using a framework such as MacZoop- all that donkey work is done for you.

How it works

When a window needs to be drawn, the Mac toolbox detects this and generates an update event. MacZoop retrieves the event and determines which window object is involved. It then asks the window object to redraw itself. The basic ZWindow takes care of the housekeeping tasks connected with handling an update, but then calls the DrawContent() method. Because this is a virtual method, the actual code that is called at runtime can be different for different windows, so although the code up to that point is shared with all windows, for our particular window is different from standard. Thus our code is called, and we draw the text. Note that when we wrote this code, we were not concerned with how and when the code is called- we can just trust that it will be. This is another new thing that programming with frameworks introduces- you generally write small fragments of code, and insert them where you know the framework will call them. You have to trust the framework will do the right thing, and in return, it trusts your code to do the right thing. It's all about mutual cooperation!

Tutorial part deux- extending your knowledge.

This part is optional. If you feel you've got enough out of the tutorial already, then please feel free to move on. However, here I will show you how to get a lot further with only a few minutes' work, so is worth doing if you have the time!

In part one, we subclassed ZWindow in order to draw our window contents. Let's go back and see what else we can do without too much effort.

Our goals for this section are:

• To make the window scrollable, and work correctly in the way we'd expect for a scroller
• Draw some more interesting graphics in the window
• To allow the Print command to operate for our window

MacZoop comes with a powerful object, ZWindow, that handles a great deal of the donkey-work of implementing a Macintosh application for you. However, the one thing it can't do, which is a very common requirement, is to scroll. Scrollable windows are very useful, and are covered in detail in another chapter. However, without getting too bogged down in how they work, let's make our ZHelloWindow scroll to show how straightforward that is.

Go back and open the ZHelloWindow.h file you created in part 1. We need to make some changes to this so that instead of subclassing ZWindow, we subclass ZScroller, MacZoop's scroling window class.

Change your header file so it looks like this:

```
#pragma once

#ifndef __ZHELLOWINDOW__
#define __ZHELLOWINDOW__

#include   "ZScroller.h"

class ZHelloWindow    : public ZScroller
{
public:
      ZHelloWindow( ZCommander* aBoss, const short id );

      virtual void    DrawContent();
      virtual void    InitZWindow();
};

#endif
```

As you can see, there are only a few changes- we include ZScroller.h rather than ZWindow.h, we base our class on ZScroller instead of ZWindow, and we added another method- InitZWindow. Save the file, then open the ZHelloWindow.cpp file you created in part one. We need to update this so that the constructor of ZScroller is called, not ZWindow, and we need to implement the InitZWindow method.

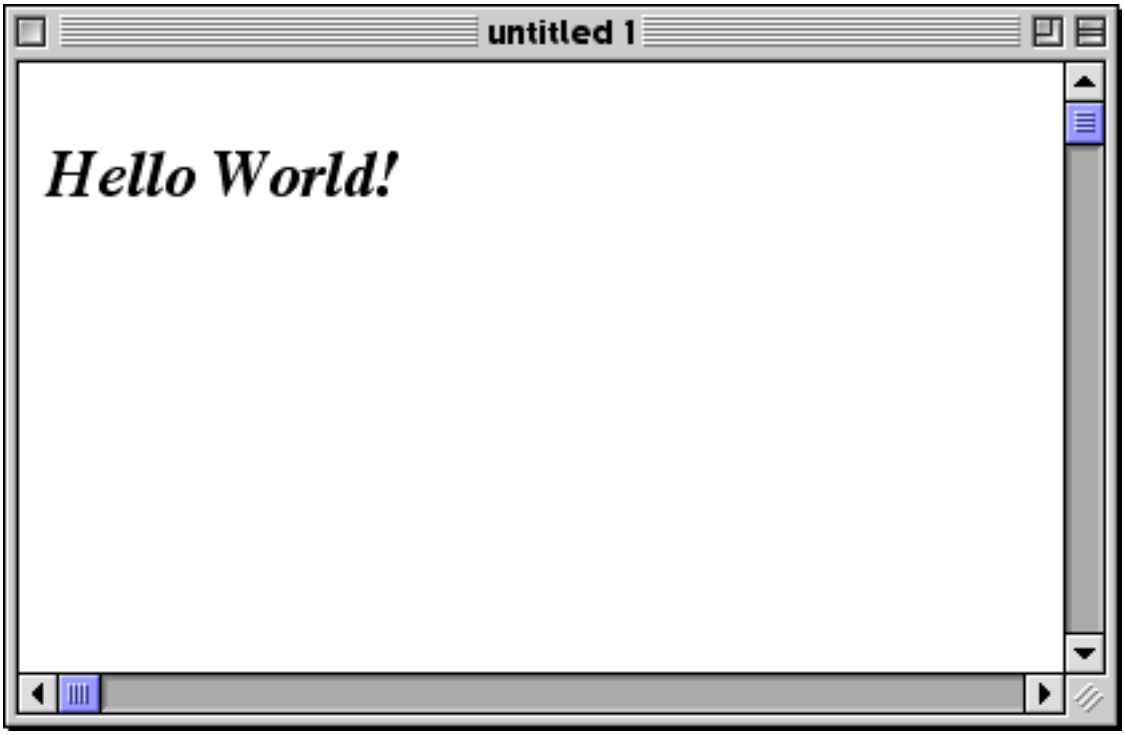
```
ZHelloWindow::ZHelloWindow( ZCommander* aBoss, const short id )
      : ZScroller( aBoss, id )
{
}

void ZHelloWindow::InitZWindow()
{
      ZScroller::InitZWindow();

      Rect r;

      SetRect( &r, 0, 0, 480, 700 );
      SetBounds( r );
}
```

There. The constructor is not very interesting- pretty much what you'd expect. The new InitZWindow method is something new- what's going on here? Well, the InitZWindow() method is a very important method of all window objects- it's where the window itself is built and set up. The framework will call this method very soon after making the window object so that the window can be initialised. What we've done here is to OVERRIDE Zscroller's initialisation method, with our own. However, we don't want to completely stop the initialisation of ZScroller- in fact we want that to happen AS WELL. That's why, the first thing we do in our new version of the method, is to call the old one. We know that will do all the correct initialisation of the window for us- we don't want or need to do that ourselves, so we call the INHERITED method. This technique is very common- we override a method not to replace it, but to extend it.

The next part of the method creates a rectangle variable, which is a common-or-garden QuickDraw structure, and sets it to 480 pixels wide by 700 pixels tall. The actual figure is not important here, we choose this to be big enough to show how scrolling works, that's all. We then call the SetBounds() method, passing the rectangle. SetBounds() is a method of ZScroller which sets up the basic scrolling area. OK, we're done.

Save the file. Now, if you've based the tutorial on the basic project, you may have noticed that while ZWindow is part of every MacZoop project, ZScroller is not- it's optional. So we need to add ZScroller to our project. To do this, either use the Add Files... command in the Project menu, or find the file in the Finder and drag and drop it into the project. The file you need, ZScroller.cpp, is found in More Classes -> Window Classes. Once done, you should be able to compile and run the modified project.

What you should see is:



This time, your window has scrollbars at the bottom and right edges. Go ahead and click them- your window scrolls! Note how the text is drawn as it is scrolled in and out of the window's view. You didn't change anything to do with drawing, yet it all works just fine. Resize the window- the scrollbars do the right thing. Magic!

If you've ever programmed the Mac before using C or Pascal, and only used the toolbox routines to get things done, you may have come across just how difficult it is to implement scrolling windows- the Mac toolbox does not go out of its way to provide a high-level approach to this very common task. So, this illustrates very well another great advantage of frameworks and using object-oriented techniques- MacZoop handles the chores for you, you just provide the content- the interesting bit.

Super Graphics!

Now, before we look at printing, let's extend our graphics a bit. Hello World is a bit boring, so we'll put some more stuff in there to make it more interesting. You can skip this part if you wish.

This time, we're going to add some stuff to our DrawContent method, so open up the ZHelloWindow.cpp file, and change DrawContent like this:

```
void ZHelloWindow::DrawContent()
{
     RGBForeColor( &gBlack );

     TextFont( kFontIDTimes );
     TextSize( 24 );
     TextFace( bold + italic );

     MoveTo( 10, 50 );
     DrawString( "\pHello World!" );

     Rect temp;

     SetRect( &temp, 100, 100, 200, 200 );
     RGBForeColor( &gRed );
     PaintRect( &temp );

     InsetRect( &temp, 20, 20 );
     RGBForeColor( &gBlue );
     PaintOval( &temp );
}
```

Now save, compile and run your application. All we've done is use standard QuickDraw calls to create simple coloured shapes in our window. Scrolling still works correctly (you may see some flickering as you scroll- this is normal, and due to the fact that we haven't bothered to optimise the graphics at all). You may want to experiment with other graphics here- feel free! Tip: If you want to find out the extent of the scrollable area of the window, use the GetBounds() method to return it.

Printing

Most applications not only want to output graphics and text to the screen, but when required, to the connected printer. This is the last thing we are going to look at in the tutorial.

MacZoop has support for printing built in to every window, so supporting the Print command is actually pretty easy. However, it's an optional part of the framework, so we need to add a bit more code to our project. Using the Add Files... command in the Project menu, or dragging and dropping from the Finder, locate the file ZPrinter.cpp and add it to the project. You'll find this in More Classes -> Printing Support.

Next, open the Project Settings.h file. On line 127 or thereabouts, you'll find the line

```
#define     PRINTING_ON    OFF
```

change the 'OFF' to 'ON'. Tip: double-click the word 'OFF' to highlight it, then type 'ON'.

Is this enough? Not quite. By default, each window is set to be non-printable. In order to print a window, you have to explicitly enable printing for that window. Since we want all ZHelloWindows to be printable, we can do that in our contructor. Open the ZHelloWindow.cpp file, and change the constructor function like this:

```
ZHelloWindow::ZHelloWindow( ZCommander* aBoss, const short id )
     : ZScroller( aBoss, id )
{
     printable = TRUE;
}
```

Now we can compile and run our application. This time, you should notice that the Page Setup and Print commands in the File menu are no longer greyed out as they were before, but can be chosen (I'm assuming that your Mac is set up with a printer- if not, make sure you use the Chooser to set one up). Page Setup does the expected thing, and by choosing Print, you should see your lovely graphics come gracefully out of the printer! Now, wasn't that painless?

Where to go from here

If you haven't already done so, it's worth getting the demo project to compile and run, since it will give you confidence that nearly every part of MacZoop is working. If you're happy, then you are ready to use MacZoop in anger. The rest of this manual discusses the most important classes and what they do, and every class is described in the Class Reference towards the back half of the manual.

Personally, I find that the best way to learn something like a framework is to be goal-oriented. Decide what you want to create, then ask yourself "what do I need to get that done?". Read up on the classes you think will help you, and don't be afraid to experiment. If you get stuck, try something else. Don't be too ambitious- make each goal quite small and achieveable to avoid frustration, as you learn new things and the concepts start to sink in, new goals and possibilities will open up. Be prepared to write of your first couple of efforts as learning experiences- invariably as you learn the framework, more efficient ways to do things will become apparent that we not so obvious earlier on. In fact, programming with MacZoop should be reasonably easy- after a while. If you find yourself having to program your way in a very long-winded fashion around something, you're probably doing it wrong- but don't worry, it's all good stuff for learning. Good Luck!

How MacZoop's commanders get things done.

In this chapter, we'll see how a very important aspect of the overall design of MacZoop works. This is the way objects fit together to implement the application architecture known as the command chain. You need to understand how this works in order to make effective use of MacZoop to make real world applications.

What is a command?

A command is an instruction from the user to the program to do something. It may be an action, such as "send this eMail", or it may be a state change, such as "switch to the pencil tool". Commands very often originate from menus. In fact, this is the main way that commands are generated in MacZoop, and in most applications in fact. However, commands can come from other places. A click on a button, perhaps in a toolbar or palette, or even from an apple event or data coming over the network.

Context

Every command has a context. In a well-designed application, the context is both obvious to the user, yet not really all that explicit- the application does what the user expects. Or it should do- this is the art of good interface design. MacZoop works to establish this context, but the details are left to you when you create your application. The main context for any command is the active window. That is the primary focus of the user's attention, and she may reasonably expect that any commands that are available will apply to that window. Within the active window, there may be other objects that further refine the context, for example a selected icon, or a piece of highlighted text. Again the user may reasonably expect that any available commands apply to the selection. Within a dialog box, an object such as an edit field may be selected- commands such as copy and paste will apply to that object, and so forth. The context is very important, and good applications will strive to make the context clear at all times. If the context isn't clear, commands may have unexpected results, and the user will rapidly become confused and frustrated with your program. Thus MacZoop helps you make a good design by reinforcing this idea of command context at all times.

Menu Commands

Menu commands are the usual way to see what commands are available, and to send them when desired. The context is reinforced by greying out commands that do not fit the current context. The user can still see the commands, so the interface is nice and stable, but the context is subtly reinforced. MacZoop makes managing the context of menus very easy to do.

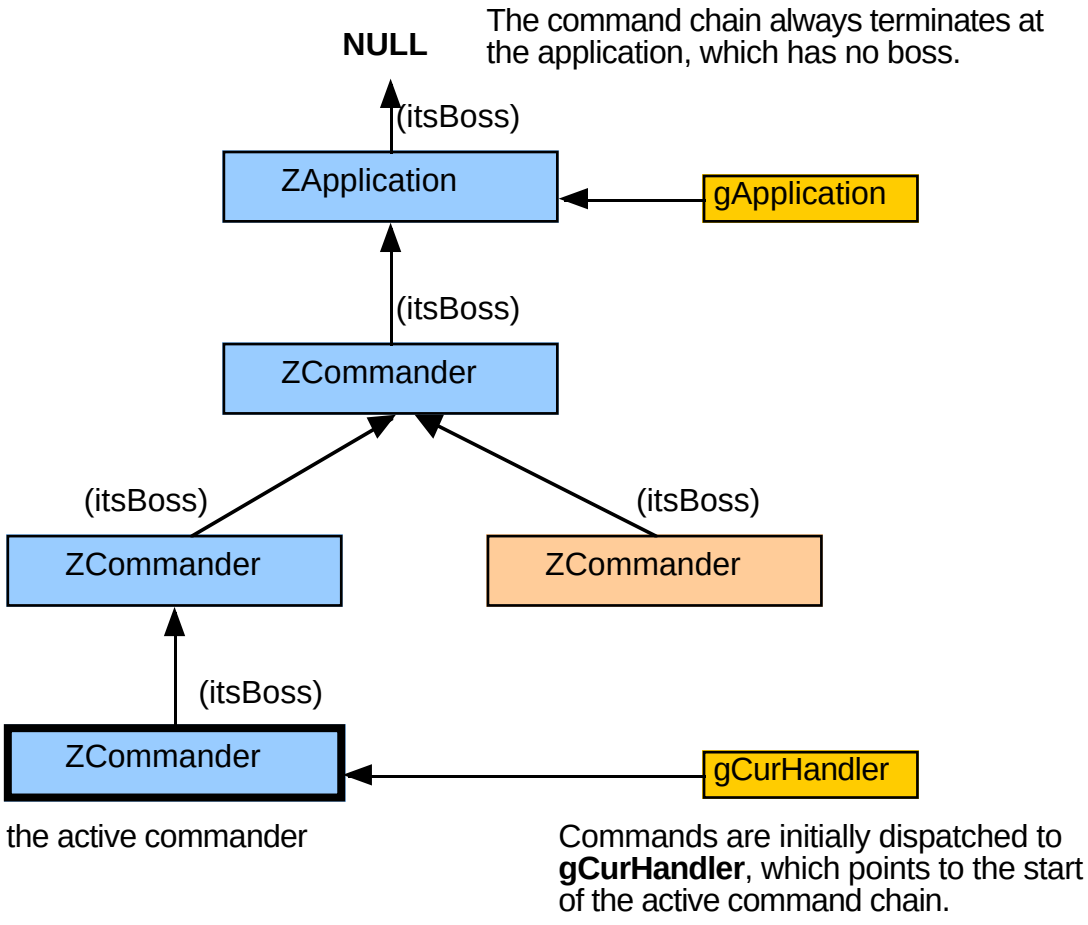
Context Hierarchy

Contexts are hierarchical. A selection is contained within the active window, which itself is owned by the current application. This goes further- the current application is frontmost and its title is displayed in the menu bar (System 8). Contexts can change quite easily- the user makes another selection, or activates a different window, or even switches to a different application altogther. Within an application, the Mac only has one active window at a time (ignoring palettes for a moment), but there may be other hierarchies at work that are less obvious. A spell-checker window may actually apply to a text window behind it, for example, so the hierarchy may be deeper than selection->window->application, it may be selection->window->window->...->ap-

plication. While MacZoop supports such hierarchies as deep as you want them, bear in mind that too much of this sort of thing can be confusing, since the user only sees one active window at once.

The command chain

The command chain is a natural extension of the idea of the context hierarchy. The command is sent to the first object that has the context, i.e. the selection. If that object knows what to do with the command, the command is handled, and that's the end of it. If the object can't handle the command itself, perhaps the next item that has the context can- the command passes up to the window. If the window can handle the command, it does so. If not, the next item that has the context, the application (or another window) is given the chance to handle it. Finally, if the application can't handle the command, it is discarded. Note that applications do not pass commands to the next level, which would be the system as a whole. An object that forms a link in the chain of command is called a Commander, and MacZoop defines a class, ZCommander, that does just this. ZCommander is the basis for a number of very important objects in the MacZoop. The two most obvious are ZApplication, the application itself, and ZWindow, which implements every window you see on the screen. Another kind of commander object are items in a dialog box. Every commander that is created must have another commander above it in the chain, in order to handle commands that it can't deal with. The exception to this is ZApplication, since we know that if this can't handle a command, the command is discarded. That's an important point- ZApplication is the ONLY commander that does not have another commander above it. In MacZoop terminology, we call the commanders above any particular commander the "boss" of the commander. Let's look at that diagramatically:



At any one time, only one commander can begin the chain of command- commands have to go somewhere initially, so somebody somewhere starts the chain. The commander that starts the chain is called the current handler. Handler is just a different word for a commander- it can handle commands. The chain is managed automatically by MacZoop, so you can largely ignore it. However, commanders you create must be ready to handle the commands they know about, and pass on the rest.

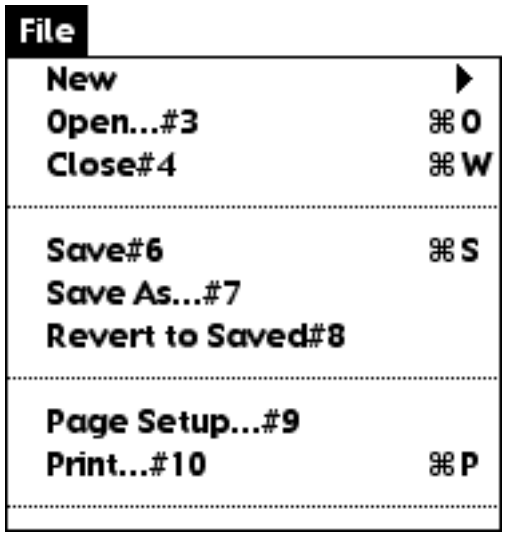
Commands

What do commands themselves look like? Well, in fact they are just a number. In MacZoop, every command is a unique 32-bit (long) number. When programming, we usually don't deal with the number itself, but declare a name to stand for the number that describes what it does in a bit more detail. This just helps make our code more readable. MacZoop itself defines and handles quite a few commands. All of the standard menu commands such as New, Open, Close, Save, Quit and so on are already defined. In fact, so that we don't accidentally cause bugs by using the same number as a MacZoop command, we reserve the command numbers from 0 to 127 inclusive as MacZoop commands. So any commands you define should start at least at 128. It is your responsibility to ensure that your command numbers are similarly unique. The commands are usually enumerated to help ensure this.

Menu Commands

Since Menus are a very common way to generate commands, we'd like to be able to attach the command number to the menu item. When the menu item is selected, the associated command number is sent off up the command chain. Unfortunately, when the Mac was first invented, Apple were not too clear about object-oriented programming and these command and context hierarchies that seem fairly obvious to us today. So much so that they forgot to provide space in a menu to store the command number. There a a number of solutions to this, and MacZoop supports two of the most common ones.

The TCL method



In the TCL method, the command number is appended to the menu item's text following a '#' symbol. When the menu is read in, MacZoop notices the #, and converts the text into the right number. The text is then removed from the menu itself, so the user doesn't see it. This is what the File menu looks like in ResEdit.

The MacApp method

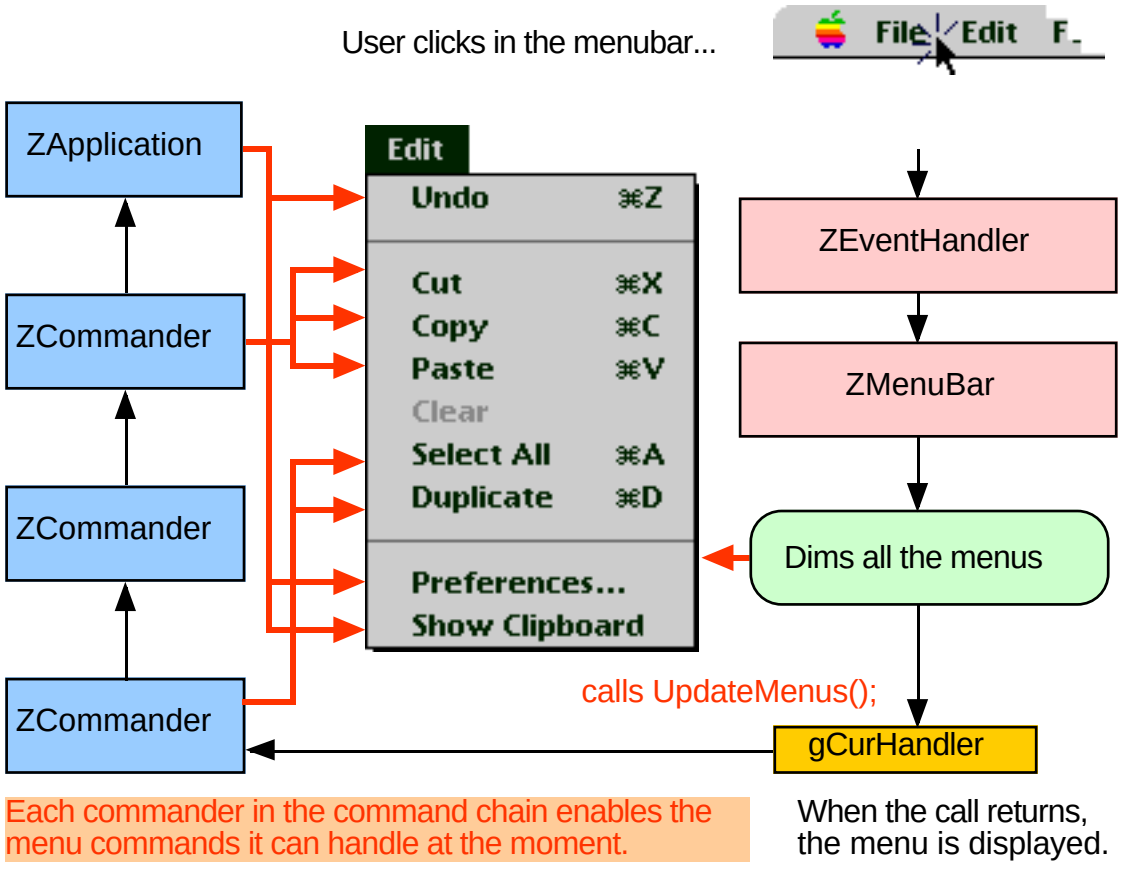
In the MacApp method, Apple recognised their earlier omission, and created a new resource format for menus, called the 'CMNU'. (Normal menus are described by 'MENU' resources). CMNU resources have space for the command number, so you can simply enter the command you want. MacZoop first looks for CMNU resources, and if it doesn't find one, it looks for the MENU resource. Either way, its taken care of- your command numbers are set up. When the user chooses a menu item, MacZoop looks up the command number for the item, then sends it up the command chain.

Note that in earlier times, it was common to program the Mac as if the menu ID and the item ID itself were the command number. MacZoop still supports this, but it has a major disadvantage. If you get half way through programming and realise that you want to move your menus around a bit, to make them more logical perhaps, or to add an item that you forgot to put in earlier, you have to go through your code and change all of the references to the menu item, since this number changes with the position of the item. With command numbers, the position is not important, so you are free to change your menus around at any time, which is good news if like me your applications tend to grow as you program!

Maintaining the context

So how do we ensure that only those menu commands that are correct for the context are the only ones available? Well, it's easy. When the user clicks in the menubar, the computer knows that the next most likely thing that will happen is that a menu is pulled down. So MacZoop jumps in and immediately disables every command. Then, since the command chain is already set up, it sends a message up the chain asking each commander to enable the commands it is prepared to deal with right now. Since only those objects that form the chain are called, commands that require a different context remain disabled. This happens so fast that the user does not notice any delay when the menubar is clicked before the menu appears, yet when it does so, only the correct commands are available. Magic!

This diagram summarises the whole process:



The method that is called to maintain the menu context is called UpdateMenus(). Most commanders will override this to enable their own commands for their current internal state, and call the inherited method to pass on the rest.

Here's a typical piece of code in UpdateMenus:

```
void MyCommander::UpdateMenus()
{
      gMenuBar->EnableCommand( kMyCommandSendMail );
      gMenuBar->EnableCommand( kMyCommandSetPencil );

      ZCommander::UpdateMenus();
}
```

The object that manages the menubar and deals with commands here is implemented by the class ZMenuBar, and the global instance of it that we talk to is referred to by gMenuBar. So every UpdateMenus() method usually has this form, calling gMenuBar->EnableCommand(<whatever>); After we've enable our own commands, we simply call the inherited UpdateMenus() to continue processing.

We can do more than just enable commands in UpdateMenus(). If the text of the menu item changes according to some internal state, this is place to change it. If you want to add a checkmark next to a prticular item, this is the place to do it. MacZoop also removes checkmarks from menus before they're pulled down, so you only need to worry about the items that should be checked. not those that are unchecked. You can also conditionally enable (or check or rename) any command according to state data you know about as you wish, so it's not uncommon to see various if(...) { .....} statements within UpdateMenus().

Here are the most common ZMenuBar methods you will call:

EnableCommand(<theCommand>)	enables the menu item associated with the command
CheckCommand(<theCommand>, <state>)	set a checkmark next to the command
SetCommandText(<theCommand>, <theText>)	set the menu text as given

Handling Commands

Commanders respond to commands by overriding the HandleCommand() method. This usually takes the form of a switch() statement, calling other methods to handle each individual command. Th edefault case represents commands that are not handled by this commander, and the inherited method should be called for these. Here's a typical example:

```
void MyCommander::HandleCommand( const long cmd )
{
     switch ( cmd )
     {
          case kMyCommandSendMail:
               DoSendMail();
               break;

          case kMyCommandSetPencil:
               fDrawingTool = kPencil;
               break;

          default:
               ZCommander::HandleCommand( cmd );
               break;
     }
}
```

This should be pretty self-explanatory. The first command calls a method to do its work. The second merely sets a data member of the class, since it's not an action, just a mode change. Other commands are passed to the inherited method. That's all there is to it.

Edit commands, Cut, Copy, Paste and Clear.

These commands are very common. So much so that MacZoop takes the handling of these commands one step further, making them even easier to implement. Since any commander could potentially respond to these edit commands, ZCommander dispatches them to the overridable methods DoCut(), DoCopy(), DoPaste() and DoClear(). You can then override these as you need to actually handle the commands. Note that DoCut() calls DoCopy(), then DoClear(), so you usually can get this command for free by implementing the other two.

MacZoop also helps to handle the Paste command context for you. Pasting not only depends on whether the object that has the context can accept Paste, but also on the type of data that is on the clipboard at the time. Therefore ZCommander gives you an opportunity to check that the data type is valid for you by calling the method CanPasteType() from its UpdateMenus() method. By default, this returns TRUE without doing any checks, but you can override it and query the clipboard's data type to see if it's something you can deal with. If you're interested in doing this, be sure to check out ZClipboard, an object that manages the clipboard and makes this trivially easy.

Numberless commands

Sometimes, it is not very feasible to assign a specific command number to a menu item. Consider a menu of fonts. Since these are usuallyl built dynamically, and the fonts will vary from system to system, it's not terribly practical to assign a specific command to a particular font. There are other instances where a menu is built dynamically, and commands are not so useful. MacZoop is happy to let you have no commands at all associated with a menu item. In tis case, you still want to get notified if the user chooses one of these menu items, so what do we do? Well, there's another form of the HandleCommand() method, that takes a pair of shorts instead of the long command number. If you override this version of the method, you'll be passed the menu ID and the item ID of the menu, as long as there was no command associated. ZMenuBar knows not to dim menu items that have no command, so you can do whatever you want in your UpdateMenus() method for these items. It's as easy as that.

Other command sources

Commands don't just come from menus. They can come from apple events, or anywhere in fact. No problem- as long as we pass commands and command like events up the comand chain from bottom to top, we'll be fine. Typing on the keyboard, for example, is a bit like a command- usually typing goes to the selection in the active window. Keyboard events are passed to the Type() method of the current handler. usually, just the current handler responds to typing, butit doesn't have to- an example is a dialog. The current handler may be an edit field, in which case most typoing results in characters being eneterd into the field, but type the TAB key, and the character is passed up the next level, the dialog, which causes the edit field selected to be switched.

Apple Events are passed up the command chain through the HandleAppleEvent() method. Like HandleCommand(), the commander responds to apple events it knows about, and passes others up the chain. While MacZoop supports only the four required apple events, registering and responding to others as you wish is pretty straightforward.

Forming the command chain

As we said, the command chain is maintained automatically as the user change sthe context, but we need a way to establish the hierarchy initially. Every commander needs a boss, and specifying whose the boss when we create our commander objects is all we need. That's why every commander object has a <itsBoss> parameter as part of its constructor. he only commander without a boss is ZApplication, but we don't make that ourselves, so we don't have to worry about it.

Windows are usually created with the application as their boss, so it's common to pass <gApplication> when creating windows. We don't have to- a window can be the boss of another window if required. This is discussed in depth in the ZWindow chapter. It is common for the boss of a dialog to be a window, if the dialog only affects things in that window. It's also common for the application to be the boss of a dialog. Dialogs themselves are the boss of the dialog items within them.

If you have a window, you can represent items within it as a commander, if this makes sense. For example, a Finder-like interface might subclass ZCommander to represent a file with an icon. Since MacZoop doesn't know how the details of your particular window are implemented internally, it provides a mechanism for you to tell it about any commanders within your window, so it can maintain the command chain. This is done using the GetHandler() method. Normally this returns the object itself, but if you further define the command chain to items within your window using ZCommander, you should override this method and return the current commander.

The Big Cheese- gApplication

Top of the command chain is ZApplication. This commander object represents the application as a whole. Every MacZoop application has exactly one object for this job, and it is assigned to the global variable <gApplication>. The job of this object is to set up the application environment, create all the various other objects that make the application work, such as ZWindowManager, and then repeatedly handle events. It is also the place where application-wide commands, such as New or Quit, are handled. That's why ZApplication is based on ZCommander. It is reasonable to want to be able to extend the repertoire of these application-wide commands. For example, you might want to open a global preferences dialog for the application, and don't care what other window is active. To do this, you need to subclass ZApplication, then extend its HandleCommand() and UpdateMenus() methods, just like any commander. However, how do we make sure that our application object is what gets created when the application starts up? After all, this object is made for us. ProjectSettings to the rescue. In the tutorial, we saw how a new window type could be substituted for the basic one by adding a definition to the ProjectSettings.h file. It's the same principle here. On line 42 (or nearby) is the line:

```
#define         APP_CLASS_NAME       ZApplication
```

If we've defined a new application class, say ZMySpecialApplication, and we've been careful to name its files ZMySpecialApplication.h and ZMySpecialApplication.cpp, all we need to do is to substitute this name into the above definition. Recompile, and MacZoop is subtly altered to make our application object instead.

Multiple Window Types

One common reason to subclass ZApplication is to provide more than one document window type. The mechanism that the tutorial explores- using ProjectSettings to set up the document window type, is fine if we only have one type, but quite often we require our application to open different types of files into different types of windows. To achieve this, we must subclass ZApplication and override the MakeNewWindowType() method. The sole purpose of this method is to create window objects on demand for a given file type. To understand it, let's look at what happens when the user chooses the Open... command.

Processing the Open... menu command

ZApplication first displays a file picker, either using Navigation Services, or the traditional Standard File dialog. The user picks a file (ZApplication determines what files to display here by examining the 'open' resource, or rather an internal copy of it, which can be extended programatically). The MakeNewWindowType() method is called, passed the file type. The correct type of window is created and initialised. Then its OpenFile() method is called, ZApplication having passed it the file to open. The window object reads the file into its internal representation, the ZApplication places and selects the window.

The usual form of the MakeNewWindowType() method is a switch statement. Here's an example:

```
ZWindow*  MyApplication::MakeNewWindowType( OSType aType )
{
     ZWindow*  zw = NULL;

     switch( aType )
     {
          default:
          case 'TEXT':
               FailNIL( zw = new ZTextWindow( this,
                         kUntitledWindowID ));
               break;

          case 'JPEG':
          case 'JFIF':
          case 'GIFf':
          case 'PICT':
               FailNIL( zw = new ZGWorldWindow( this,
                              kUntitledWindowID ));
               break;
     }

     if ( zw )
     {
          try
          {
               zw->InitZWindow();
          }
          catch( OSErr err )
          {
               ForgetObject( zw );
               throw err;
          }
     }

     return zw;
}
```

Here, we can open several file types- TEXT files cause ZTextWindows to be created, and four different graphic file types cause ZGWorldWindows to be created. Whichever type we create, we initialise it and return it. ZApplication subsequently sees to it that the file information is passed to the window.

File Handling in ZApplication

The file-handling capabilities of ZApplication are quite important to our application. While it is ZWindow that actually opens and parses files, it is ZApplication that coordinates the process, as described above. Because it is ZApplication that manages the Open... file dialog, we need a mechanism to tell it which files to display. Normally, this is set up using standard resources. System 7.0 used the classic “Bundle” resources to allow an application to tell the Finder what file types it could accept, and which icons were associated with which document types. Later, the ‘open’ resource was added for greater flexibility. MacZoop can use both or either set of resources to set up the list, plus you can alter this programmatically.

In addition to the file types list, every application has a creator code, or signature. This is also usually held in the ‘BNDL’ resource. When MacZoop starts up, the ‘BNDL’ resource with ID number 128 is checked. If present, the global <gAppSignature> is set to the application creator code. This global can be used in a number of places in MacZoop to set the creator code of files created by this application. In the classic Mac OS, the BNDL resource contained a list of ‘FREF’

resources which link particular icons to particular file types. If that is all we have, MacZoop will scan these FREF resources and build the internal file type list from them. However, if we have the more modern ‘open’ resource with ID number 128, that is used to build the internal instead.

This process is controlled by a number of settings in ProjectSettings.h. In order for it to take place at all, USE_SIGNATURE_FROM_BNDL must be ON. If this is off, the value of gAppSignature can be set by setting the value <kApplicationSignature>. (Note that if USE_SIGNATURE_FROM_BNDL is ON, yet the application has no bundle, gAppSignature will be initialised to ‘????’. The second part, the scanning of the ‘open’/’FREF’ resources, is controlled by CHECK_FREF_RESOURCE_TYPES. If this is OFF, your file type list will be empty, and only the application signature will be set from the bundle. If the list is empty, Mac-Zoop displays ALL files in the Open... dialog.

You can programatically add a file type to the internal list using ZApplication’s AddFileType() method. You can query whther a particular file type is in the internal list using the CanOpenFileType() method. The format of the internal list is private to ZApplication.

Sometimes, you want to display the Open file dialog, but with a different file list than the one that ZApplication generally uses. To do this, you need to build a file type list of the correct format, passing it to ZApplication’s PickFile() method. Since the format of this list is private, ZoopUtilities provides a simply way to create a suitably formatted list, the function NewFileTypesList(). This accepts a list of files as a count and an array of OSType values, and returns a Handle to the opaque list structure, which you can pass to PickFile(). Afterwards, you should dispose of the handle.

Note that PickFile will automatically use Navigation Services if available and turned on using the _USE_NAVIGATION_SERVICES ProjectSetting. (If compiled in, this will work correctly, using Standard File, on systems that do not have Navigation installed).

Event handling in ZApplication

ZApplication’s main job is to fetch and dispatch events as the application runs and the user interacts with it. In fact it actually passes off most of the details of this task to a helper object, ZEventHandler. You can replace the entire event handler if you need to, although this would be very unusual.
The application spends nearly all its time inside ZApplication’s Run() method. This repeatedly calls the Process1Event() method until the user Quits. Run() also set sup the execption handling for the entire application, and deals with the memory shortage fund. Process1Event() is a very important method. It fetches one event from the queue, and dispatches it. There is a version of this method that can accept events fetched externally, and process them using the usual dis-patcher. This can be useful if you employ a third-party library which supplies events. Apple’s Navigation Services is an example of this.

Since ZEventHandler actually does all of the actual work, if you are interested in how events are handled in mor edetail, please refer to the class reference for that object.

Application phases.

There are three basic phases in ZApplication- initialisation, running, and quitting. You can deter-mine the current phase using the GetPhase() method. While most time is spent in the running

phase, it is likely in many cases that you'll want to do various things during the initialisation phase. You have a number of opetions. The recommended method is StartUp()- this is called relatively late in the initialisation phase, so most things are set up by then. In general, you should avoid doing any initialisation except setting values into data members in the application constructor. This is because this is called very early, in fact befor ethe Mac toolbox is initialised, severely limiting what you can do here.

While StartUp() is a good place to do many things, the common requirement of creating your 'fixed' or global windows should not be done here. The reason is that if there is a problem creating your window, the application will not be able to start up- exceptuions at this point are always fatal. This may be OK if your window is vital, but in many cases, it's better to try to carry on with limited functionality. To do this, another method, RunFirstTask() is provided. This will be called exactly once atthe very end of the initialisation phase, but in fact from within the Run() method. This gives it the full protection of the execption handling mechanism. Usually you should create your fixed or global windows here.

Quitting

To quit the application, call RequestQuit(). This called when the user chooses the Quit menu command or the quit apple event is received. Note that it is only a request- the applicaiton will quit as long as the user doesn't stop it- a quit can be cancelled if the "Save changes?" dialog appears for an unsaved window while quitting, and the user hits cancel. So be prepared for the quit phase to return to the running phase.

You can get notified when the application is going to quit by overriding the ShutDown() method. Once this is called, the application WILL quit. You cannot attempt to force it not to here, or subsequently very bad things will happen.

Getting Information

You can obtain lots of information from ZApplication. Process information such as the process serial number, the filespec of the application itself and its name are all easily obtained using the GetProcessInformation() and related methods. You can get the name of the application using GetName(). Refer to the Class Reference for a full description of all the available methods.

ZWindow: At the heart of MacZoop

ZWindow is possibly the most important class within the MacZoop framework. This chapter describes the class and explains how it is used.

What is a window?

This may seem like a dumb question, but surprisingly, windows are often misunderstood. A window is a way to share a limited system resource- the monitor- among any number of clients. Windows effectively multiply up the available monitor space because they can overlap and scroll. Therefore the most important function of a window is to provide a graphics environment where graphical objects and text can be drawn to the screen. The Macintosh toolbox provides a set of data structures and functions that implement this aspect of windows very comprehensively and effectively. MacZoop does not interfere too much with this side of things, since the QuickDraw graphics routines and the basic structures of windows are effective solutions to providing graphical output.

Windows are much more than simply places to do graphics, however. A window is a primary object that the user interacts with while using the computer. By selecting a particular window, bringing it to the front and making it active, you are telling the computer a great deal about your expectations and the sort of work you want to do. The graphics provide an output to the user, but in addition, windows should be able to respond to the user's input, in the form of mouse clicks, drags, menu selections and typing on the keyboard. The Mac toolbox is less complete when looking at the input side of things at the window level, usually requiring that you pick up the various user-generated events yourself and direct them to the appropriate window. MacZoop handles all of that for you, so when programming with MacZoop, you can forget about how these events are resolved, and simply treat the window as the input device AND the output device.

In many application designs, a window is used to represent the primary view of some data (sometimes called the DATA MODEL), and this data may be stored in a disk file when the application is not running or when the user is working on something else. Associating a window with a file is often referred to as a DOCUMENT window. All windows in MacZoop are, by default, document windows, in that they can have an associated file. You don't have to use this feature if you don't want to, but if you do, it's all there and ready to go.

Taken together, the behaviours of windows within an application, including the document interface, can add up to a fairly complex programming task- dealing with windows as they are selected and deselected by the user, directing input to the active window, drawing the contents of the window, responding to updates, tracking the state of a document and handling common actions such as Save and Revert. You'll be pleased to hear that MacZoop deals with all of this for you. All of the classical standard behaviours that every Macintosh application should have are built-in and will "just work". As the programmer you are free to concentrate on what your window will actually do that differs from the default, rather than havng to worry about all this infrastructure.

The downside is that having all this functionality concentrated in one place makes the API for ZWindow look rather daunting at first glance.

Overview of MacZoop windows

A MacZoop window is defined by the class ZWindow. This class inherits from ZCommander, so a window can respond to menu commands, keyboard input, etc. ZCommander inherits from ZComrade, so a window can send and receive messages to any other MacZoop object derived from ZComrade. ZComrade inherits from ZObject, so windows can be streamed (if this feature is turned on- see streaming).

ZWindow brings a number of new abilities to the picture. It can:

- Provide a place to draw graphics and text
- Respond to mouse clicks
- Respond to dragging and dropping from other windows or the desktop
- Respond to the user typing on the keyboard
- Associate a file on disk with an arbitrary data model
- Handle standard commands Save, Save As, Revert, Close, Cut, Paste, etc.

ZWindow, or one of its standard derivatives such as ZScroller, will be the class you will mostly be working with when programming real applications with MacZoop. Naturally, you will be making a subclass of ZWindow to actually make a specific kind of window that does something useful for your application.

ZWindowManager

MacZoop organises its windows on screen with the use of a global helper class called ZWindowManager. This object is automatically created, and is responsible for dealing with the ordering and placing of windows on screen, and keeping the floating window layer separate from the normal windows. You may sometimes call upon ZWindowManager to carry out some function for you, but you will very rarely need to subclass it, or even care about how it works. Any windows that are created must be known to the window manager however. Normally, this will be taken care of for you, but bear this in mind if you are doing something unusual.

ZWindow Basics

There are three steps required to put a window on the screen. These are:

1. Create the window object
2. Initialise it
3. Make it available to the user

Let’s see how this is done.

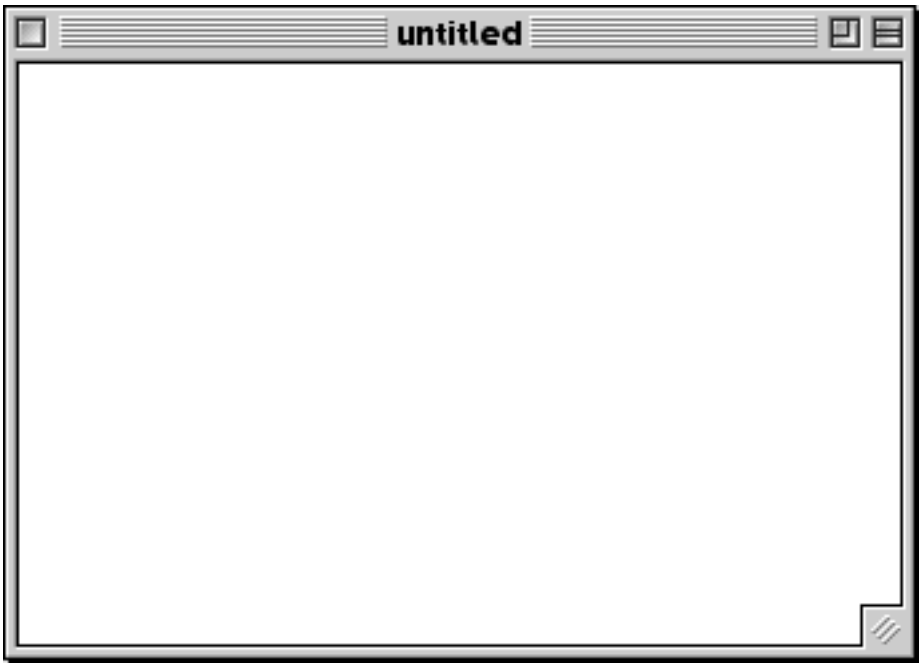
```
ZWindow*    w;

FailNIL( w = new ZWindow( gApplication, kMyWindowID

try
{
      w->InitZWindow();
      w->Select();
}
catch( OSErr err )
{
      ForgetObject( w );

      throw err;
}
```

The object is created using the new operator, as you might expect. Note that we wrap this up in a FailNIL in case there wasn’t enough memory. After creating the object, the next step is to call its InitZWindow()

method. This step is absolutely vital- if you forget to do this, your window will be a dead duck, and will most likely crash your system. If this is so vital, why isn't it done as part of the object construction? The answer is that different types of window have different initialisation requirements, and having the separate method makes it possible to have a single call for all window types by using the dynamic polymorphism of C++. Just don't forget to call it! Note that the code uses a try/catch block to trap any errors at the initialisation stage and discard the window if so. You should do the same.

Finally, the window is made available to the user by calling its Select() method. This makes the window visible and active. This step does not have to be done immediately- you can wait until a later point before showing the window. Also, you don't have to make it active, you can simply show it instead, or use the ZWindowManager method MoveWindowBehind() to place it behind another window.

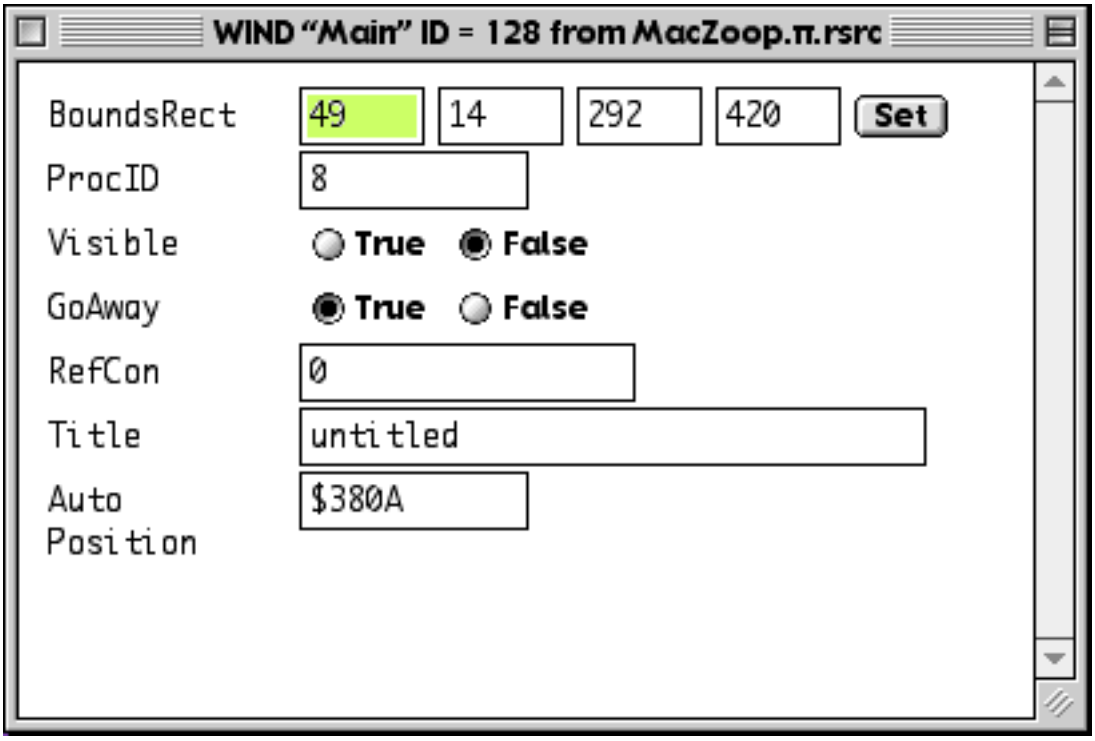
You should now have a window that looks like this:



As you can see, it's not terribly exciting, but it has all the essential qualities you need. If your window looks different, it may be because you have System 7.x, in which windows look a little different, or perhaps because the template resource was set up differently. Let's look at the window template now.

WIND resources

All MacZoop windows are created from a WIND resource (except dialogs, but that's dealt with in the next chapter). This is a simple template that describes the basic features of a window. When you create the ZWindow object, you pass the resource ID of the WIND resource that is used to set it up. Here's the WIND resource as shown in ResEdit:



(Note that normally ResEdit shows the window graphically- you can see the template as above by using the Open As Template menu).

<**boundsRect**> is the default size and position for the window, inglobal coordinates. In fact, MacZoop usually repositions the window automatically for you, so only the sizes of the sides are important.

<**procID**> is the window type. It is easier to set this up using the graphical editor in ResEdit, which allows you to set the type by clicking on an icon representingthe window features. However, some types, such asthe newer floating window types, require this to be entered manually as ResEdit does not directly support them (at the time of writing). This number reflects whether the window has a zoom box, for example, or a grow box.

<**visible**> should always be FALSE for MacZoop windows. This is important- if you forget and your windows are initially visible, you may experience some odd behaviour in your MacZoop application.

<**GoAway**> sets whether the window has a close box or not. Generally, they should.

<**refCon**> is not used. Note: MacZoop uses the refCon field internally- it is not available for application use.

<**title**> is the window's title. Note that MacZoop automatically deals with making sure the titles of multiple windows are unique, as per the Human Interface Guidelines, by appending 1, 2, etc to the title as multiple windows with the same title are created.

<**Auto Position**> is generally ignored, since MacZoop dynamically positions the window for you.

Once you have created a window, the next step is to learn how to draw graphics there.

Drawing in Windows

Drawing is primarily done in response to an "update event". This is generated whenever the OS detects that part of a window needs to be redrawn- for example when it is brought to the front from behind another window, or dragged on screen from partially offscreen or whatever. Whenever part of a window gets clobbered for whatever reason, an update event occurs. MacZoop preprocesses this event to the point where all you have to do is actually draw. You do this in your DrawContent() method.

By the time DrawContent() is called:

- The window is the current port
- The clip region is set to the window's interior
- Any scroll offset is adjusted for (for scroller windows)

MacZoop also takes care of drawing any controls that you have attached to the window.

When you draw, you refer to the coordinate system of the window. This is always set so that the top, left corner of the interior of the window is at 0, 0 for ZWindow. Drawing is simply a matter of making the relevant QuickDraw calls. You do not have to worry about drawing over the edges of the window, etc.

Example:

```
void  MyWindow::DrawContent()
{
      Rect  r;

      GetContentRect( &r );
      InsetRect( &r, 20, 20 );
      PenSize( 4, 4 );
      RGBForeColor( &gRed );

      FrameRect( &r );

      InsetRect( &r, 4, 4 );
      FrameOval( &r );
}
```

Sometimes you'll want to use the clipping region yourself to limit drawing to particular areas. You can do this, but it is good technique to form the union of the original clip region and any new one, just in case there are areas (such asthe phantom scrollbar areas) that you don't want to draw into. Here's how to do that (this uses the global utility variable gUtilRgn):

```
void  MyWindow::DrawContent()
{
      Rect        r, rc;
      RgnHandle   sClip;

      sClip = NewRgn();
      GetClip( sClip );

      GetContentRect( &r );
      InsetRect( &r, 20, 20 );
      PenSize( 4, 4 );
      RGBForeColor( &gRed );

      rc = r;
      rc.right = ( r.left + r.right ) / 2;
      RectRgn( &rc, gUtilRgn );
      UnionRgn( sClip, gUtilRgn, gUtilRgn );
      SetClip( gUtilRgn );

      FrameRect( &r );

      SetClip( sClip );
      DisposeRgn( sClip );
}
```

99% of the time, just doing your drawing in DrawContent() will be fine. Your code may be called at any time, even when the window is inactive (background windows still get updates!), so it may wish to to query the IsActive() method if what is drawn changes between the active and inactive state. Very occasionally, waiting for the update event to be generated and processed may not be fast enough. In this case, there are ways to bypass some of the processing to varying degrees to speed things up. Let's suppose you make a change to your data model and wish to reflect the changes in your visualisation. The simplest way is to call PostRefresh(). This posts an update for the content area of the window, and causes an update event, which is processed as normal. When you DrawContent() method is called, your code picks up the change in the data model, and renders its view accordingly. This method is very reliable, and generally fast. You should use it where you can.

Faster update processing may be accomplished by posting the refresh as normal, but then immediately calling the PerformUpdate() method. This responds as if the update event had occurred, but without actually fetching the event. It is faster because the update event is processed without giving up any time to background applications, or executing toolbox code to create the event. This technique is also very compatible.

Even that may be insufficiently fast for your needs. (Are you sure?) Finally, it's possible to call the drawing methods directly. Doing this is becoming less compatible, since the responsibility for various setup steps becomes yours. You can call Focus(), then call Draw(). Focus() makes sure the port is set up correctly, and Draw() deals with clipping the content area, calling DrawContent() and drawing the controls. Finally, if you're already set up and want the fastest possible drawing, you can call DrawContent() directly, or even a method of your own that it would normally call. Taking this route can be fraught with unexpected problems, so always see if one of the more compatible techniques will work sufficiently well first.

More Drawing Tips

By and large, DrawContent() is yours- you can change the graphics environment to suit yourself and draw. When this method is called, the previous port state may not have been preserved- it usually will have, but you cannot guarantee it. Therefore it's a good idea to establish the port state up front, for example by calling PenNormal(), TextFont(), etc, etc as needed. The utility function SetPortBlackWhite() is also handy for setting the basic port drawing colours to black and white.

If you are concerned that the changes you make to the port's state are causing problems elsewhere, there is a simple solution. For example, your window consists of two panels, one which has bold text, and a second with plain text. Each is drawn by a separate method, both called by DrawContent(). To ensure that the bold setting of the first does not carry over to the second, you can use a helper object called ZGrafState. This object records the settings of the port before any changes are made, then afterwards, it puts them back exactly as they were before. Normally, this object is used as a stack object, so it is self creating and deleting, which makes it trivially easy to use.

```
#include    "ZGrafState.h"

void  MyWindow::DrawContent()
{
      Str255      s = "\pHello World!"
      Rect        r;
      ZGrafState  gs;

      GetContentRect( &r );
      InsetRect( &r, 20, 20 );
      PenSize( 4, 4 );
      RGBForeColor( &gRed );

      FrameRect( &r );

      InsetRect( &r, 6, 6 );
      TextFont( kFontIDTimes );
      TextFace( bold + italic );
      TextSize( 18 );

      TETextBox( &s[1], s[0], &r, teJustLeft );
}
```

Note that although the colour, pen and text parameters are all changed by the code, because <gs> is automatically disposed when the method goes out of scope, all the original port parameters are put back as they were before.

Dos and Don'ts of DrawContent()

Rule 1. Accept that the current port is correct and draw to it.
Rule 2. Do not assume that <macWindow> is the current port.
Rule 3. Do not change the origin with SetOrigin() in this method.
Rule 4. If you change the clip region, always sect it with the clip region set when the method was called.

Rule 5. Do not call Focus() or SetPort() from within DrawContent().
Rule 6. Do not assume you're the active window. If your data view changes according to the activation state, you need to keep track of this state and draw accordingly.

The reason for these rules is so that scrolling and printing work properly. In fact it is easier and less work to comply with the rules than to break them. One potential pitfall is when using the CopyBits() toolbox call to draw parts of your content. Rather than a port, this function takes bitmap parameters, so it is tempting to use <macWindow->portBits> as the destination parameter. Don't do this! While this will work for rendering to the screen, it breaks rule (2) and thus your window's contents will not print. Instead you should call GetPort() to obtain the current port, then use the pointer returned as the destination parameter. Alternatively you could use <qd.thePort> instead.

Printing

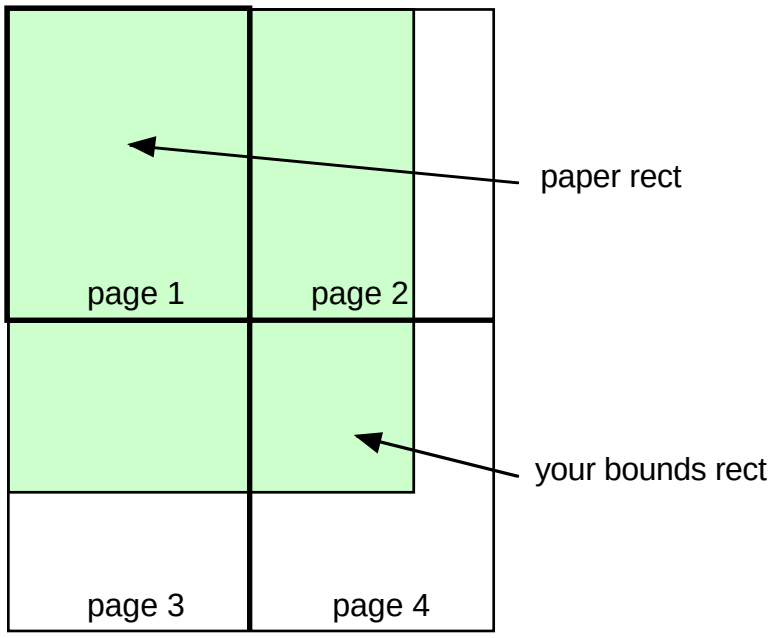
Windows automatically support printing. The code you write to draw your window content will also provide printed content without any changes needed. However, for optimum results, it is worth bearing in mind a few things.

To support printing, your window should set its <printable> member to TRUE. It defaults to FALSE. Unless you do this, the Print command will be greyed out when your window is active. Printing is handled by ZPrinter, which must be part of your project.

When printing, the <isPrinting> member is TRUE. This can be examined by your drawing code to avoid drawing things to the printer that do not make sense, such as widgets in the window, or large erased areas, etc.

Pagination is determined by the CalcPages() method. By default this paginates the window content as a series of tiles in row major order. You can overridethis for other pagination schemes. While printing, your DrawContent() will be called once for each page to be drawn, with the port origin offset as required. Your DrawContent() method should therefore never change the port origin. You can examine the clip region to see which part of the image is being called for.

If you stick to the rules outlined in the discussion of DrawContent(), you should find your window will print fine. You will rarely need to override the standard PrintOnePage() method, since this does all that is necessary to set up and call your DrawContent() method. However, you may want to override the standard pagination method CalcPages(). Here you are passed the paper size as a rect and you are expected to return the number of horizontal and vertical pages of this size needed to completely cover your bounds rect. The standard method tiles the bounds across the paper in column-major order (i.e. horizontally, then vertically). Thus your pages are numbered like this.



Note that PrintOnePage() assumes this is how the bounds is paginated. However it does not need to overridden if you just want to force a 1-column or 1-row pagination, since you just need to return 1 in the appropriate parameter in CalcPages(). You only need to override PrintOnePage() if you desire a completely different pagination method.

PrintOnePage() is called as many times as needed- once for each page. It calls DrawContent() so you may want to check the area being actually printed to avoid drawing stuff that is not on the given page. In this case, your content rectangle is totally irrelevant- it does not matter where the view is scrolled to when printing. Instead you need to look at the bounding rectangle of the current clipRgn to determine what part of your bounds is being printed.

Before starting a print job, the window's PrintingStarting() method is called. Afterwards the PrintingFinishing() method is called. By default these methods merely set and clear the <isPrinting> flag. You can override them to do other stuff if you want.

User Input to ZWindow

Because ZWindow is a subclass of ZCommander, it can respond to typing, apple events and menu commands that are passed up the command chain. When a window is active, it (or a nominated object within it) will begin the command chain. A window can respond to those commands that it wants to handle, and pass the rest to its boss. A window's boss may be the application or another window- its boss is passed as a parameter when the window is created.

As for any commander, ZWindow enables menu commands in the UpdateMenus() method, then calls the inherited method to update menus "above". It overrides the HandleCommand() method to respond to commands, passing others on to the inherited method. Edit menu operations are handled by overrides to the DoCut(), DoCopy(), DoPaste(), etc methods as for ZCommander.

The basic ZWindow object handles the commands Close, Save, SaveAs and Revert. More on this when we discuss the document methods.

Typing on the keyboard results in a stream of calls to the Type() method, once per character typed. The default method does nothing, but you can override it to respond to the keyboard as appropriate.

Mouse Clicks

Mouse input is possibly the most important input method to a window. To receive mouse input, your window class overrides the Click() method. This is passed the location of the click and any modifiers. You can and should respond directly- all prefiltering to make sure the right window is targetted has been done. By default, ZWindow calls any attached modifiers with a click message, then calls a default ZMouseTracker (if this feature enebled). It is normal to override this method however. Note that ZScroller already overrides this method to distinguish and handle clicks in the scrollbars and the content area. For a ZScroller or derivative, you should respond to the mouse by overriding ClickContent(), and not Click().

You Click() method is permitted to keep control in a modal loop if required, for example while doing a drag. However, it should return as soon as the mouse is released. You can also use a ZMouseTracker object to deal with dragging and selection in a window, of which more later.

Here's a typical mouse drag loop:

```
void  MyWindow::Click( const Point mp, const short modifiers )
{
      Point       curr, old;
      Rect        dr;

      curr = old = mp;

      PenMode( patXor );
      PenPat( &qd.gray );
      PenSize( 2, 2 );

      Pt2Rect( &dr, curr, mp );

      while( WaitMouseUp())
      {
            GetMouse( &curr );

            if ( DeltaPoint( curr, old ))
            {
                  FrameRect( &dr );
                  Pt2Rect( &dr, curr, mp );
                  FrameRect( &dr );
                  old = curr;
            }
      }
      FrameRect( &dr );
      PenNormal();
}
```

Handling the Cursor

Quite often, the cursor shape changes as it moves over different parts of your window. To implement this, override the AdjustCursor() method, and check where the mouse is, then call SetCursorShape() as required to change the shape. This function call will use colour cursors if available. You also get passed the modifier keys, so if your cursor changes shape according to the modifiers, you can do that here also.

Double-clicks

To respond to a double-click is quite easy. Basically, you query gApplication->GetClicks() in your Click method. This is set to the number of clicks, 1 for single, 2 for double, 3 for triple, etc. For a click to be detected as a double or more, several criteria must be met.

1. The clicks must have been in the same window
2. The clicks must have occurred within the system's double-click time
3. The clicks must have been in the "same place".

What constitutes the "same place" is up to you- you overried ZWindow's ClickInSamePlace() method to provide a definition. By default, this just returns TRUE. But you decide whether the click positions are in the same place in whatever way you wish. You are passed two consecutive positions, and must return TRUE or FALSE. Note that MacZoop avoids trying to claim that

clicks within, say 4 pixels are in the same place, as MS Windows does. What is logically the same place varies- by implementing this properly, you can make your application much easier and more intuitive to use.

ZWindow and the document interface

ZWindow can associate a file on disk with the window. This is the basis behind the document interface. Normally, you'll represent the data internally in some way (you don't have to, you can simply read it from the file as needed). You window is responsible for converting between the three forms of the data model- disk<-->internal<-->visualisation. There is no reason why your internal data model cannot be embodied in another object- it's up to you.

When the use chooses the Open... command from the File menu, the application object will deal with displaying the Standard File or Navigation dialog and choosing the file. It is also responsible for creating the correct type of window for that type of data, since you'll have overridden its MakeNewWindowType() method to do so (see ZApplication). Once the window has been created and initialised (all done by ZApplication), the window itself is called to read the file and set up the internal data model. This is done in the OpenFile() method.

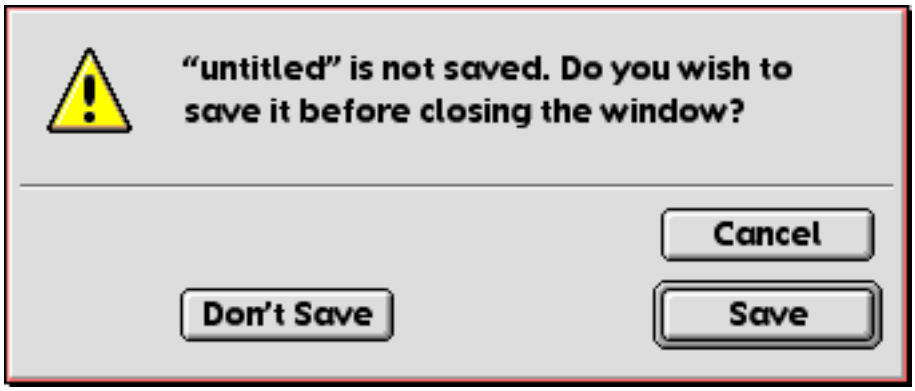
Your OpenFile() method is passed the file type of the selected file and a flag to say if it's a stationery file or not. In addition, your SetFile() method was already called which sets the <macFile> data member to the FSSpec of the chosen file. The default OpenFile() method does not open the file- it merely sets the window's title to the file name and resets some internal flags. Your window will override this method to actually read the file, and then call the inherited method to do the housekeeping.

What you do here is up to you. Often, you will use a ZFile object to handle the actual reading of data from the disk. This can be a generic ZFile which reads bytes and your window understands the format, or a subclass of ZFile which reads and writes in a particular format for your window. Which way you prefer to organise it is up to you.

Once the file is read, the window is displayed on screen, an update generated, and the content refreshed accordingly. This step is done for you by ZApplication, calling your window as usual to do the drawing.

Tracking document state

Often, your window will not just be a way to visualise your data model, but also away to edit it. Input fropm the keyboard, menu commands and the mouse can all be used to change the state of the data model in some way. So that the user is correctly prompted to save changes when this happens, you can inform the window about state changes in your data model. You do this by calling SetDirty() with a parameter of TRUE, whenever the state changes. If the user tries to close the window while thie dirty flag is set, they'll see the familar "Save Changes" alert:

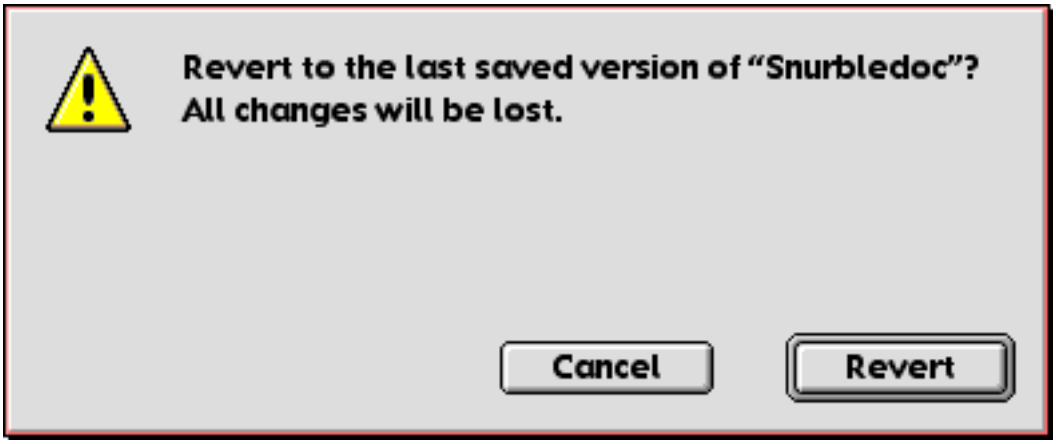


ZWindow will respond to the user's choice by routing to the appropriate method as required. Again, this standard behaviour comes without any programming effort on your part- you can simply concentrate on the one little piece of the puzzle you need to provide, namely, writing the data model to the file. This is done in the SaveFile() method.

Like OpenFile(), exactly how you do it is up to you, but you should call the inherited method to do the housekeeping at the end. This includes updating the window title if you renamed the file, etc.

Revert

If you design your OpenFile() method so that it can be called more than once without ill effect (usually pretty easy!), then the Revert() command should just work. You are asked to confirm the Revert:



And if so, OpenFile() is called again, and the requirted updates posted, etc. If you need to do something different to get the revert to work, you can override the Revert() method instead if you like.

If you have appearance and have compiled this option in, MacZoop will also manage the window title proxy icon for your file also, and handle dragging of this icon to the Finder.

Saving Files

The user can choose Save or Save As at any time. (Save is only available if the doc is dirty and the file really does exist on the disk, in other words, if <macFile> is valid). ZWindow will display the Stsndard filew dialog or Navigation window for saving a file, and allow the user to set a title and location for the file. Then SaveFile() is called as before to actually write the file to disk.

Thus, the complete document interface is implement using just two overrides- OpenFile() and SaveFile().

Undo

MacZoop supports Undo. While this is not specifically a window feature, ZWindow provides some of this support. Instead of simply setting the dirty flag, you can alternatively create an undo task. This is an object derived from ZUndoTask which records enough information to restore your data model to its previous state. What this means is up to you, it may contain a complete copy of your data model (feasible only if memory requirements are modest), or be more sophisticated. In any case, the undo task itself is responsible for carrying out the work needed to alter the model. Once created, you pass it to ZWindow's SetTask() method. This both dirties the document and works with ZApplication to handle the undo task and manage the Undo menu command.

Window sizing and placement

Windows can be placed anywhere on screen, and the user can grow or shrink them, zoom them, move them around, etc. By and large, all of this will just happen without any coding. However, you will probably want to have some control over this.

Window growing

When the user grows or shrinks a window, there is usually a minimum and a maximum size that is sensible for that partcular window. MacZoop sets this up very easily. The ZWindow method SetSizeRect() is the basis of this. Here, you pass a rectangle, whose top and left fields dictate the minimum size of the window, and whose bottom and right fields set the maximum size. Other code in MacZoop ensures that the window will always conform to these limits- the user will not be able to make the window bigger than the max or smaller than the min. The limits are honoured for stacking, tiling and zooming operations too.

Calling SetSizeRect() may be inconvenient. If so, this can also be set using a resource of type 'WLIM'. When a window is craeted, a 'WLIM' resource is looked for with the same ID as the WIND. If found, the value of SetSizeRect() is set accordingly. A WLIM resource is simply a rectangle, as passed to SetSizeRect().

Placement

Before a window is shown, it may be placed. (If not, the original position as given by the WIND resource is used). Normally, ZApplication calls Place() for windows it creates. The default ZWindow method Place() passes the buck to ZWindowManager to placethe window using its InitiallyPlace() method. This stacks windows from the top left of the screen as they are created. In addition, if the window is being opened as a document with an associated file, the position of

the window can be read in from a resource in the file automatically.

Other placement options are PlaceAt() which allows a window to be positioned at any specific location in global coordinates. This is called by ZWindowManager's InitiallyPlace() method once it's worked out where to place the window. It's also possible to place a window with respect to another already visible window by calling the PlaceRelative() method. Here, you pass a WindowPlacing constant which tells ZWindow how to do the placement.

Zooming

Zooming allows the user to flip the window between two states- the user stae, which is how they set it, and the standard state, which is determined automatically. The standard state is intended to make the window as large as necessary to show all of the window contents, but no larger. However, it is constrained by both the maximum sizerect and the size of the monitor it is displayed on. MacZoop deals with all of this automatically. The only piece of information that the programmer needs to provide is the size of the area representing the data. Even this is unnecessary in the case of a scroller, since the scrollable area is usually considered the maximum data area. The method GetIdealWindowZoomSize() is the place where this information is provided- override it if you have something special in mind.

Position Saving and Restoring

MacZoop handles this common chore for you. However, to obtain this behaviour you should turn on the _WPOS_WINDOW_PLACEMENT compilation option in ProjectSettings.h. This saves the window position in the window's file, if it has one, or else in the prefs file, if not. Although restoring is done for you when Place() is called, saving is not. The reason is that the programmer need to have control over this, since in not every case will the position want to be saved. The programmer also needs to supply an ID under which to save the position, since the original WIND id is not sufficient (many windows are often created from a single template). To save the position, call SavePosition() with the id you wish, or 0 to use the WIND id.

Drag and Drop- Dragging into a window

All windows have the ability to support drag and drop, right out of the box. By and large, the mechanics of this are all done for you, so unless you have special requirements, you only need to override one method to handle a drop, and that is Drop(). This is called as many times as needed when items are dropped on your window. The default method does nothing, but the unpacked data is all passed in as parameters.

ZWindow does not accept all drops without question. You can set it up only to accept certain data flavours. This is done by overriding the AcceptsFlavour() method. For each flavour passed in, you return true or false according to whether you can handle it. Only if you return TRUE for at least one flavour for any particular drag will the window hilite and accept the drop.

If you need finer control over the drag and drop process, you can override at a finer level. There are methods for every step of the process, from DragHandler(), which dispatches the basic drag to the methods EnteredWindow(), EnteredHandler(), InWindow(), LeftWindow() and LeftHandler(), and the hiliting method DragHilite(). For a drop, you can override DropHandler() if the default unpacking and passing items individually to Drop() is inadequate.

Note- it's often possible to combine clipboard handling with drag and drop, so your override to Drop() may be able to call DoPaste(), or both methods can call a common handler.

Dragging from the window

To drag items from the window requires a little more work on your part, since ZWindow has no knowledge of what your window contains. ZWindow handles the basic mechanics of a drag, but you have to set up the drag data. To start a drag, you call the Drag() method, passing in the mouse point. MacZoop deals with the coordinate conversion for you, but calls two methods to set up the drag- first, MakeDragData(), in which you should create drag data in the correct format using Drag Manager functions, then it calls MakeDragRgn(), which is where you should build the drag outline. ZWindow provides a default drag outline which is the window content rect, which may be sufficient.

Window Closure

In MacZoop, closing a window is usually the same as disposing of it. Windows dispose of themselves, and because of this, windows must be heap objects, never stack objects. A window is closed when the user clicks the close bos, or the Close menu command is chosen. In both cases, the Close() method is called. The first thing this does is to close any child windows that may be open. You will recall that it is possible for a window to have another as its boss. If this is the case, then when the boss is closed, any windows subsidiary to it must be closed first. This is done by calling those window's Close() methods, so complete hierarchies may be closed in one call. A subsidiary (child) windo may refuse to close (the user may Cancel the Save Changes alert, for example). If so, the original Close() method returns without further processing.

Next step is to examine the dirty flag and see if changes need saving. If so, the user gets the Save Changes alert. Again, if the user cancels this, no further processing is done. If the user elects to save the changes, the Save() method is called to deal with it.

Finally, if the window is really closing, it is first hidden, then deleted. If closed, Close() returns TRUE, else FALSE. Once Close() returns TRUE, any further references to the window are stale and should not be used.

Sometimes, you do not want to delete a window when closed- it might only be a temporary closure. In this case, it is common to override Close() and simply call Hide() instead. In this case, you can delete the window yourself, but the Save Changes alert will be bypassed.

Autoclose

ZApplication implements the command Close All, when the user selects the command together with the option key. This iterates through all the open windows and closes them. Sometimes, you may want this command to overlook a particular window. To do this, simply set the data member <disableAutoClose> to TRUE (default is FALSE). Your window can still be closed if explicitly commanded to do so, but not by the Close All command.

Floaters

MacZoop supports floating windows of all types. For a window to float, its <floating> data member must be set to TRUE before the window manager is passed the window initially. This is

usually done automatically by the MakeMacWindow() method, which examines the window definition to see if a floating style is being created. Thus, to get a floating window, all one needs to do is to give it a floating appearance in ResEdit. MacZoop provides the correct behaviour.

n.b. while you can set <floating> manually, you are advised not to. Users will get confused if the window appearance of a window does not agree with its behaviour.

As well as the newer system floating styles, the common Infinity Windoid is also supported.

In every other respect, floaters behave like any other window. However, since they are never "active" in the usual sense, they will not be part of the command chain and will therefore not respond to commands or typing unless some other object (e.g. the application) takes deliberate steps to pass on these events. For this reason, you are strongly discouraged from putting editable text fields into a floating window, since a) they are unsupported by default, and b) the target of the keyboard will rarely be obvious to the user. Floaters are fine for palettes, etc that can be manipulated using the mouse only.

Summary of common ZWindow overrides for user events

Event description	To respond, you override:
Update event for any reason	DrawContent()
Cursor moved over window	AdjustCursor()
Mouse click in window	Click()
Activate (window comes to front)	Activate()
Deactivate (window no longer in front)	Deactivate()
Keyboard input	Type()
Menu update	UpdateMenus()
Menu command	HandleCommand()
User closes window	Close()
User drags into window	Drop()
Determining what is acceptable in drag	AcceptsFlavour()
User drags out of the window	Drag()
Resolving double-clicks	ClickInSamePlace()
User opens a file	OpenFile()
User saves a file	SaveFile()
User clicks in the zoom box	GetIdealWindowZoomSize() (if required)
User resized the window	WindowResized()

Window methods by functional category.

This is not a complete list, but are the most commonly used methods for a variety of purposes. For a complete list of methods, refer to the class reference.

Description	Method to call
1. Construction & set-up	
Initialise any window object	InitZWindow()
Establish size limits for window	SetSizeRect()
Place window according to rules	Place()

Place window in global space	PlaceAt()
Place window relative to another	PlaceRelative()
Set the size of the window	SetSize()
Create the underlying Mac structures	MakeMacWindow()

2. Drawing output

Set the window as current port	Focus()
Post an update for the window's interior	PostRefresh()
Handle an update immediately	PerformUpdate()

3. Window State

To hide the window	Hide()
To show the window	Show()
To show (if necessary) and bring to front	Select()
To Close	Close()
To set the window's title	SetTitle()

4. Document handling

To mark as dirty	SetDirty()
To mark as dirty using an undo task	SetTask()
To save the window's position in its file	SavePosition()
Is the window dirty?	IsDirty()

5. Misc info about the window

Is the window in front and active?	IsActive()
Is the window printable?	IsPrintable()
Is the window visible?	IsVisible()
Is the window a floater?	IsFloating()
Is the window resizeable?	IsResizeable()

6. Getters

Get the window's title	GetName()
Get the interior rect	GetContentRect()
Get the height of the title bar	GetTitleBarHeight()
Get the structure region	GetStructureRegion()
Get the content region	GetContentRegion()
Get the widths of the window's borders	GetStructureFrameBorder()
Get the location of the window on screen	GetGlobalPosition()
Get the Macintosh windowPtr of the window	GetMacWindow()
Get the associated file spec	GetFileSpec()
Get the associated file's type	GetFileType()
Get the background colour of the window	GetBackColour()

7. Printing

Calculate pagination for printer output	CalcPages()

Window assistants

There are one or two additional classes which may help you work with windows. First, remember ZWndow is a Commander, so you may attach modifiers to it. ZWindow sends out modifiers messages for draw, click, open and close. Thus it is possible to contain your drawing and clicking code in a ZModifier object rather than subclassing ZWindow, if this makes sense.

ZMouseTracker

ZMouseTracker is an object that embodies the basic behaviours required for handling mouse drags in a window. Of course, you can use the code given earlier to handle mouse drags, but ZMouseTracker has numerous advantages. First, it uses much more sophisticated drawing techniques to provide flicker-free animation (if you tried the code example, you may find it's quite flickery, and varies with the CPU speed of your Mac). Second, it passes tracking information back to the window using the standard ZComrade messaging methods. Third, it can tell if it's being used in a scroller and autoscroll the window automatically. Fourth, it can automatically "grid" the drag to particular grid dimensions and limit the drag area overall.

To use a mouse tracker, you create one in your Click() method (or use the default behaviour provided by ZWindow), then respond to the messages it sends back to your ReceiveMessage() method. You could also of course subclass ZMouseTracker and implement drag responses there if you wish.

ZMouseTracker messages:

msgMouseTrackStarting	the drag has just started
msgMouseTrackNewPosition	the drag has moved to a new position
msgMouseTrackComplete	the drag is complete
msgMouseTrackScrolledView	the drag auto-scrolled the view

For each message, the mouse position is passed back in the <msgData> parameter. (Pointer to a Point structure).

Typical scenario: suppose your window has an array of selectable icons. The user can select icons by dragging a box in the window- icons that are within the box or touch its edge are selected, all others are not. To implement this, you respond to the msgMouseTrackNewPosition message by either calculating the rectangle of the drag, or simply asking the mouse tracker itself to tell you what it is. This is done using its GetDragRegion() method, which returns a region, but you can look at the region's bounding box to get the rect. Alternatively, you could simply keep the rect yourself. Then you scan through your list of icons, and see which are within the box. Those that are are highlighted, those that are not are unhilighted. Now, usually changing the highlight state of an icon involves redrawing it. Where the drag outline intersects an icon, the outline will be erased. Before continuing, it is necessary to "patch up" the drag outline by calling the tracker's UpdateDragRegion() or UpdateDragRect() method. This is vital for smooth and trouble-free selection in this kind of situation.

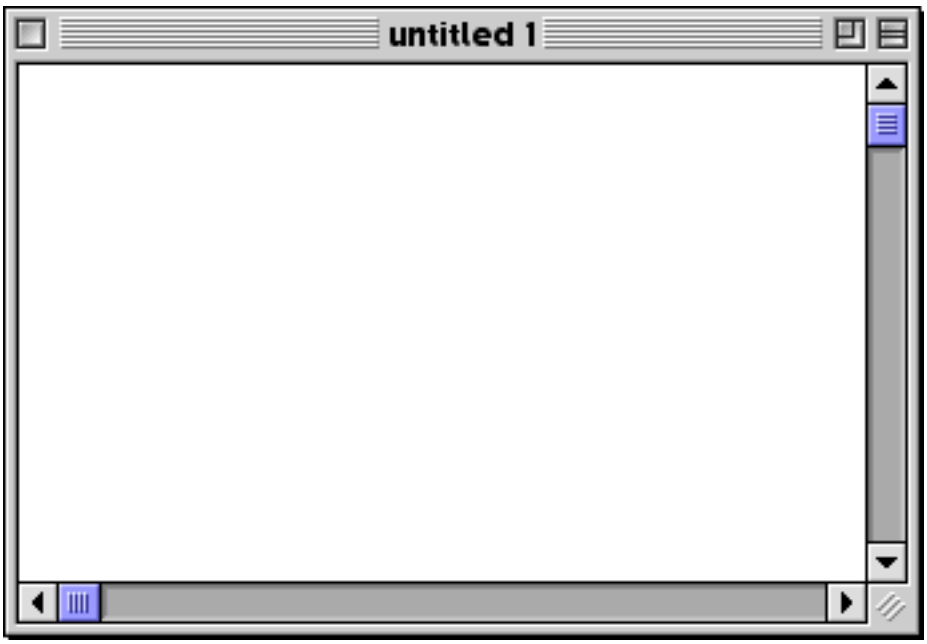
ZWindow derivatives: ZScroller

While ZWindow is crucial to anything of more than trivial complexity, it is more likely that ZScroller will be your window class of choice when it comes to creating your main user inter-

faces. ZScroller frees you from the limitations of drawing area that ZWindow has- in ZScroller, your drawing area can be very much larger than the window.

ZScroller handles the chores of the scrollbar positioning, tracking, etc for you. In general, you can simply override DrawContent() as usual, and get the expected results. However, ZScroller introduces some additional concepts which should be borne in mind while developing your application.

First, let's look at a basic scroller. It is created exactly the same as ZWindow, except that this time you call new ZScroller rather than new ZWindow. What you get will look like this:



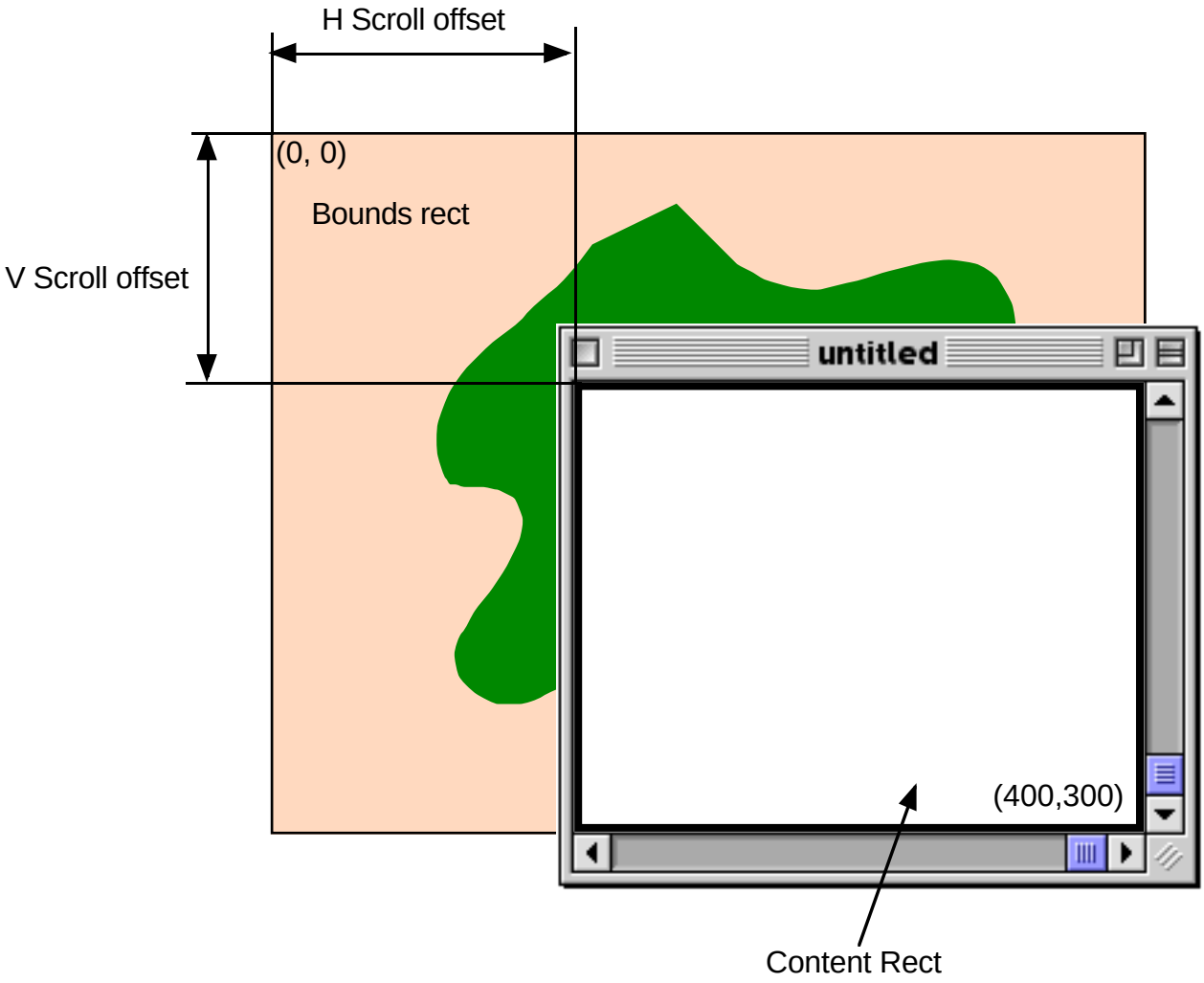
Well, actually it probably won't- we cheated here. If you use the code exactly as for ZWindow, it's likely that the scroll bars will be inactive, but for now that's no problem. If you resize the window, you'll see that the scrollbars cling to the sides as you'd expect. The Mac toolbox requires that you program this- MacZoop does not, since it deals with all of that for you.

The bounds rectangle

ZWindow introduced the concept of the Content rectangle- that's the interior part of the window that shows your content, but does not include any scrollbars, or the grow box, etc. ZScroller still has a content rect, and naturally it excludes the scrollbars. The big white area in the middle is the content rect.

With ZWindow, the content rect is all the space you have to draw in- anything else is simply clipped out. With ZScroller, you have much more drawing area, and this is called the bounds rect. The content rect merely shows part of the bounds rect- you can reveal other parts of the bounds rect by, you guessed it- scrolling.

This diagram should help illustrate the various rectangles and the relationship between them and the scrollbars.



Here, we have a bounds rect (the orange area) from 0,0 to 400, 300 in size, being viewed through a much smaller content rect, shown with a heavy border (the actual size of the content rect is unimportant). The view is scrolled such that both scrollbars are at their maximum value. The green blob represents something drawn into the bounds rect. Note that as the view is scrolled, the coordinate system of the bounds rect remains unchanged- all that changes is the relative offset between the content rect and the bounds rect. Thus when we draw, we always draw in the bounds rect coordinate system, and thus we can ignore the scroll offset.

The bounds rectangle is set by calling the SetBounds() method. The maximum size of this rectangle is the entire QuickDraw plane, which is (-32767, -32767, 32768, 32768). This is a huge area- at 72dpi, it represents well over 5000 square feet!

When drawing, you can use the GetBounds() mehod to find out your drawable area. When DrawContent() is called, the coordinate system is aligned with the bounds rect automatically. Thus you can simply draw and things will appear as expected. However, you can see that if you draw the entire image every time, even though only part of it is visible, you are wasting CPU time that could be better spent on something else. Therefore it's often useful to work out just what is visible and then only draw those parts. Let's see how it's done.

DrawContent() takes no parameters, so you need to figure out the visible area yourself. All the information is readily to hand, so it's easy. What you need to know is, what is the content rect, expressed in terms of my bounds rect? To obtain this information, simply call GetContentRect() from inside your DrawContent() method. Because the port's origin has been manipulated for

you, the coordinates are automatically transformed- magic! (Note- calling GetContentRect() from outside of the DrawContent() method, perhaps from code external to the window, will return a rect with the top, left corner at 0,0)

Now you have this information, you can simply add code to your drawing code to only draw stuff that intersects this rectangle. Note- the toolbox function SectRect() can be very useful here! Here's an example:

```
void  MyScroller::DrawContent()
{
      // draw a series of icons avoiding drawing those out of view
      // icons are stored in <theIcons> array and are objects

      TIcon*      icon;
      Rect        cr, ir;
      long        n;

      n = theIcons->CountItems();

      GetContentRect( &cr );

      while( n )
      {
            icon = theIcons->GetObject( n-- );

            icon->GetIconRect( &ir );

            // in view?

            if ( SectRect( &ir, &cr, &ir ))
                  icon->Draw();

      }
}
```

Calculating control parameters

This section is really provided for your interest and further understanding, it is not needed to use ZScroller.

The parameters of the scroll bars are their current value, their minimums and their maximums. For two scrollbars, this is six variables that need to be set up. These values are affected by the size of the window, the size of the scroll bounds, and the offset between the scroll bounds and the content rect.

Example. Suppose the scroll bounds rect is 600 pixels across. Our window is sized so that its content rect spans 250 pixels, and the view is scrolled to the far right edge. What are the control parameters?

The control minimum is equal to the left-most coordinate of the scroll bounds. This is zero in our example, but could be any value between -32767 and 32768. The distance that the scroll bounds can scroll is the difference between the span of the bounds and the span of the window. As the

window spans 250 pixels, there are 600 - 250 = 350 pixels out of view. This is therefore the control maximum. If the control is set to 350, it means that the left edge of the content rect is 350 pixels from the left edge of the bounds, leaving 250 in view. Thus the control is in the far right position.

The control parameters are all calculated by the CalculateControlParams() method of ZScroller, and this is called whenever the window size or the bounds is changed.

ZScroller calculates all of its scrolling values in terms of pixels. A change of 1 in the scrollbar's value means the view shifts 1 pixel. If the control changed only by this amount for each click on the arrow of the scrollbar, scrolling would be smooth, but very slow indeed. Therefore it's usual to shift the view more than this at a time. By default, ZScroller shifts the view 10 pixels at a time. This is set by the <hScale> and <vScale> data members. You can set these to anything convenient using the SetScrollAmount() method.

Note that these values are also used to determine the paging values for the scrollbars. This is the amount that the view will scroll when the paging areas of the bars are clicked. This is calculated as the span of the content rect less the value of the scroll amount in the appropriate direction.

It is common to set the scroll amount to the line height of a text view, or the cell height of a list view, etc, so that whole numbers of items are scrolled at once and paging is intuitively correct.

Autoscrolling

A nice feature to incorporate in order to make your applications easier to use is autoscrolling. This is where a scrollable view scrolls during a mouse drag so that you can select more than is visible in the window at once. ZScroller naturally supports this.

In order to autscroll during a modal drag, simply call the AutoScroll() method within the drag loop, passing in the current mouse point. If the point is outside the content rect, the view is scrolled accordingly. MacZoop's autoscrolling is quite intelligent- it scrolls faster the further outside the content rect the mouse moves. This gives a very fine degree of control while dragging that is sadly lacking in many applications- including, I'm sad to say, CodeWarrior, whose autoscrolling is nearly uncontrollable on faster machines. Having this degree of control gives excellent usability on a wide range of CPU types. Your users will thank you for it!

If you use ZMouseTracker with ZScroller, autoscrolling is automatic.

Mouse Input

ZScroller handles the mouse slightly differently from ZWindow. In ZWindow, you override Click() to deal with mouse events. In ZScroller, this has already been done, since ZScroller needs to deal with clicks and drags in the scrollbars themselves. For clicks and drags in your content area, there is a new method, ClickContent(). This takes the same form as Click(), but the mouse coordinates are transformed to the bounds rect, not the content rect. Thus as with drawing, your click handling is relative to the bounds rect of the view, which makes sense, I hope you'll agree!

If you don't override ClickContent(), you'll get default ZMouseTracker behaviour, if this feature is enabled.

Grabber

ZScroller also provides another common feature for free, in any scrolling view. If you command-click in the view, you can drag the view directly using a “hand grabber” type tool. ZScroller automatically manages the cursor shape for you too.

Live scrolling

So-called “live” scrolling became popular when the introduction of the PowerPC meant that there was sufficient processing power to make this work well. This means that the view scrolls as the scrollbar’s thumb is dragged, rather than springing to the new position when the ghosted outline is released. ZScroller supports live scrolling by default, though you can turn this off if you want.

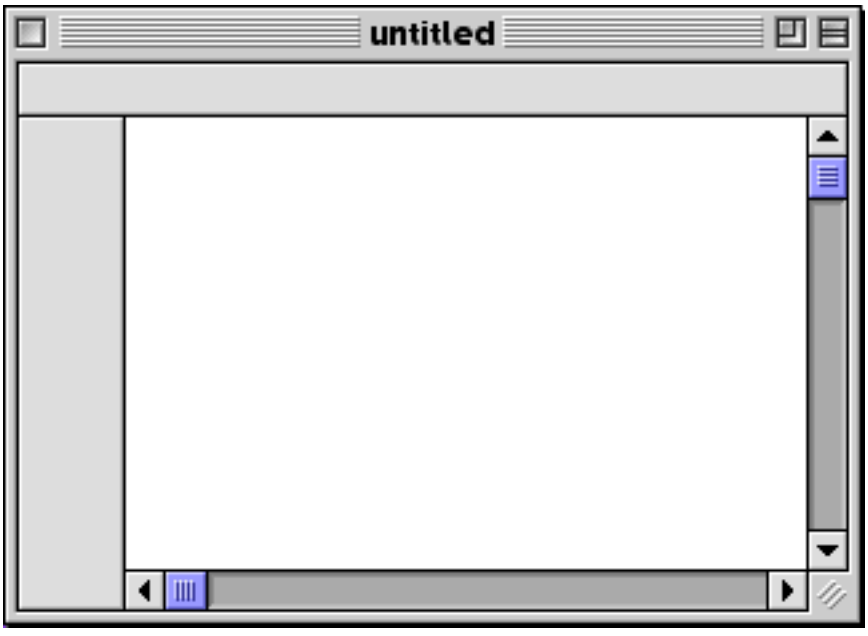
Headers and margins

ZScroller adds the ability to have separate areas for window headers and a margin on the left-perhaps for an in-window toolbar. These are provided as part of ZScroller rather than ZWindow because in ZWindow you are free to draw whetever you want in the window as you wish. In a scroller, the main area is reserved for a scrolling view, so if you want fixed header or margin areas, special provision is required.

To set a header, set the <topMargin> data member to the height of the header, and to set a left margin, set the value of <leftMargin> to the width of the margin you require. Scrolling calculations take these into account such that these areas will never scroll.

To draw these areas, override the DrawHeader() and DrawLeftMargin() methods as required, and you can respond to clicks in these areas using the ClickHeader() and ClickLeftMargin() methods. ZScroller sets up the clip regions to these areas when they are called.

By default, ZScroller predraws these areas using appearance panels if you have appearance available and turned on in your project settings.



That’s pretty much all there is to ZScroller- everything you can do with ZWindow you can do with ZScroller, but in addition you get scrolling!

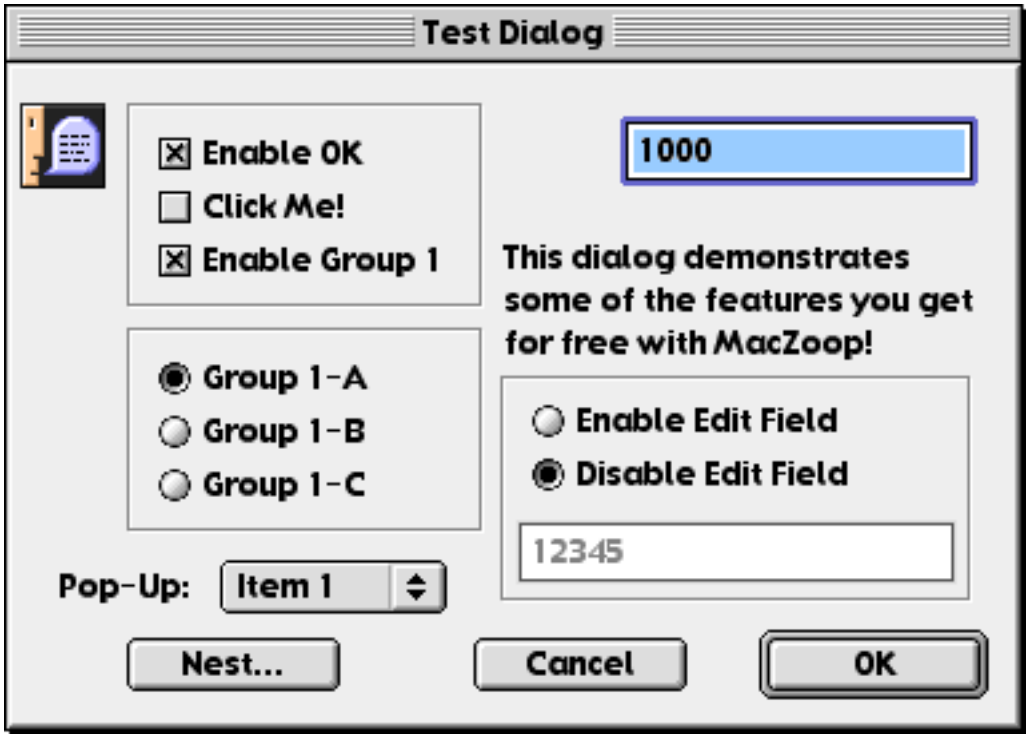
Other Available window types

MacZoop provides a number of handy window classes. These are:

- ZWindow - general class for any window or dialog
- ZDialog - general dialog box class
- ZScroller - generic scrollable window class
- ZTextWindow - styled text editor based on TextEdit
- ZPictWindow - window for viewing PICT files
- ZGWorldWindow - PICT window with double-buffering (GWorld)
- ZProgress - standard progress bar dialog
- ZLMListWindow - window with List Manager cells within
- ZInspectorWindow - window that receives messages about other window activity
- ZMDEFWindow - window that can display a menu (tear-offs, etc)

Only ZWindow is a required class for a MacZoop project, though most windows in useful applications will be based on ZScroller or other derivative window.

All About ZDialog



ZDialog is MacZoop's class for handling all types of dialog box. In common with many frameworks, MacZoop largely bypasses the Mac toolbox Dialog Manager in order to provide a more comprehensive set of services, but unlike many frameworks, MacZoop uses parts of the Dialog Manager where it makes sense in order to provide a maximally compatible programming experience. Thus the programmer can rely on familiar dialog creation techniques such as using ResEdit to create DLOG and DITL resources, while getting a much richer set of services within the code, and more complex user interfaces from the same basic tools.

ZDialog itself is a subclass of ZWindow, so ZDialog embodies all of the features of ZWindow, including having an associated file if required, etc. Within the content area of the dialog box, however, ZDialog goes further and provides an array of user-interface objects (dialog items). By managing the standard behaviours of these items, the programmer is relieved of the task of programming them individually. So far, so much like the Dialog Manager we all know and loathe. ZDialog goes a lot further however, because it implements the items as objects within the MacZoop framework. This allows new types of user-interface elements to be created much more simply, and thus extend the range of features of a dialog very easily and powerfully. Not only does MacZoop provide the capability to do this, but it also provides some actual interface objects, such as list boxes, scrollable text boxes, progress bars, scrolling icon lists, etc. All of these items can be created and set up using straightforward editing within the ResEdit DITL editor, just as any standard item can. So, lots of extra power, but no new tools to learn!

The standard item behaviours incorporated into ZDialog include managing checkboxes, which automatically flip state, radio button grouping, where selecting one button in a group automatically turns off all the others in the same group (all of which is set up in a delightfully simple way in ResEdit's DITL editor), and dimming of items and groups of items. Also, text entry fields

(called Edit Fields throughout this manual) are also extended so that standard behaviours such as filtering letters and numbers, providing hidden (password) fields and checking numeric entries for validity is all handled for you, and again simply set up using the usual DITL editor in ResEdit.

Class Overview

ZDialog is the basic dialog window object. This is a subclass of ZWindow, and can be used to create all types of dialog boxes- modal, modeless, moveable modal and even floating dialog boxes. ZDialog contains an array of its items. A dialog item is embodied in the class ZDialogItem, itself a subclass of ZCommander, and the basic class handles all of the standard types of item, such as buttons, checkboxes, icons, static and editable text fields. Only one object class is used to handle all of these types. This class is Appearance aware, and dialog items adopt the systemwide appearance. Further classes for the “extended” items are subclasses of ZDialogItem, and include ZListDialogItem for list boxes, ZTextDialogItem for scrollable text boxes, ZCPopDialogItem for a pop-up menu for picking colours, and numerous others. This list of special extensions may grow in the future as more useful user-interface items are developed, but they will all have the same basic API and not require changes to ZDialog or ZDialogItem to operate. Thus by understanding the API presented here, you will be able to apply it to all future dialog items with very little extra work.

Simple Dialogs

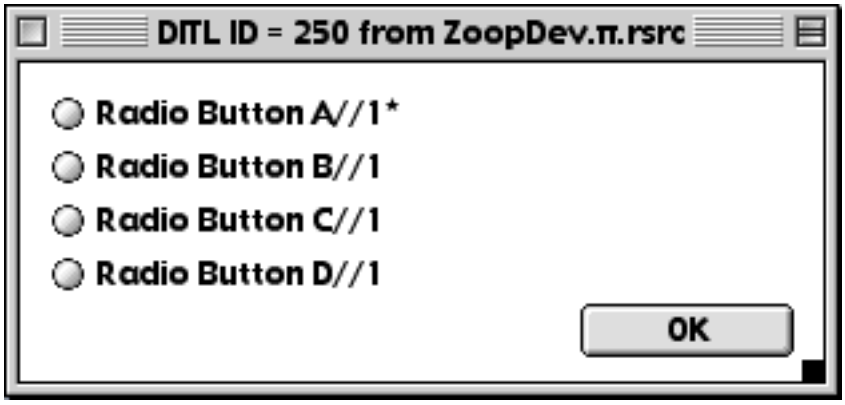
“Simple” Dialogs are those based on so-called ‘pure’ DLOG/DITL resources. MacZoop can read a completely standard template and build all of the required objects with no intervention on the part of the programmer. Doing this is virtually identical to creating a window, except that you request a ZDialog object, and pass it the ID of the DLOG template resource. Since this is so common however, and very often a subclass of ZDialog itself is not needed, ZCommander provides a utility method, OpenSubDialog(), which does all the necessary creation work and only needs the resource ID of the DLOG passed to it.

If the DLOG/DITL template contains only the standard, original Macintosh data, then the resulting dialog will have none of the standard handling features such as radio grouping. The only exception is checkboxes, which will be inverted automatically when clicked. The reason for this non-activity is to provide a fully compatible mode with the Dialog Manager, where the programmer wants to have full control over all activity. Usually however, we will prefer to get MacZoop to do all the tedious donkey-work for us. After all, if it did nothing more than the Dialog Manager, we haven’t gained much, right? In order to get a standard copy of ResEdit to do the work for us, and to avoid special resource types or non-standard stuff “hidden” inside standard types, we simply use the text strings that most items have associated with them in a variety of ways, adding extra codes and flags to them. At run-time, MacZoop parses these strings as part of the dialog build process. The original, unmodified string is then returned to the item so the user is none the wiser that anything different is going on in their resources. In fact, this method is so versatile, powerful, fully compatible and yet simple, it’s a constant mystery to me why Apple or anyone else never thought of it!

Standard Items

Radio Button Grouping

This is one of the most often requested Dialog Manager features that never was. Most Mac newbies, especially if they have experience of other platforms, are amazed that groups of radio buttons have to be programmed individually on the Mac. With MacZoop, that's a thing of the past. This ResEdit DITL template shows how it's done:



Quite simply, each button's title string is modified by appending two forward slash characters, followed by the group number, in this case 1. Valid group numbers are from 1 to 255. Group number 0 is not valid (all ungrouped buttons are assigned to group 0, but MacZoop intentionally treats them as not part of any group, for compatibility with the Dialog Manager). Naturally, when a button in a group is clicked, all others with the same group ID are turned OFF, and the clicked button is turned ON. As a further help, you can set one button in a group to be the initial choice when the dialog is created, by adding an asterisk (*) to the end of the title string, as shown above for button A. If omitted, all buttons will be initially OFF. Of course, you can programmatically set any button in the group.

Grouping for dimming

Sometimes, you need to be able to dim all items in a group at once, or re-enable them all at once. MacZoop permits this. In addition, it can often be helpful to simultaneously dim more than one group of buttons, or include buttons in a group for dimming that are not part of a set of radio buttons. MacZoop also permits this. You can add a group spec to any button, including checkboxes and standard pushbuttons. These will be ignored as part of a set of radio controls when clicked, but will be included while dimming a group. In addition, you may have more than one group of radio buttons which in themselves act independently, but you would like then to be dimmed as a set nevertheless. This is possible by using a special numbering convention for groups. For clicking, all 16 bits of the group ID is checked, actually permitting 16,383 independent groups (negative IDs are not allowed), but for dimming, only the bottom 8 bits are compared. Thus group 257 and group 1 are distinctly separate when clicking, but are considered the same group when dimming. Dimming is accomplished by calling ZDialog's EnableItem() and DisableItem() methods. If you pass a positive number, the item with that ID is individually dimmed, but a negative value is interpreted as the negative of the group ID, and all items in that group, to 8 bits significant, are dimmed or enabled.

Edit Fields

Edit fields are text entry fields. The Dialog Manager does not distinguish the data that is entered into such a field, and expects the programmer to code all of the individual cases to validate such fields. If the user types in rubbish, many programs do not make much effort to prevent it or validate it before moving on because it's just so much work. MacZoop to the rescue! You can easily specify as part of your DITL resource what a field can legally accept as it is typed in, and for a numeric entry field, specify that the field be validated between a min and max limit that you specify before the dialog is allowed to be closed. Not only that, but handling of hidden (password) fields is completely automatic, hiding the typed characters and maintaining the true data elsewhere. Another rather laborious chore is handled by the framework as standard.

Resource specification for an edit field is similar to radio button grouping. The text of the field is followed by two forward slashes, then a flags value. Two other values are also optionally appended, separated by commas. The syntactic format of an edit field is:

```
[<default text>][//<flags>[,<min value>[,<max value>]]]
```

Most important are the flags, since they specify the filtering and other general behaviours of the field according to this table:

Flag Value	Bit	Effect
2	1	Signed Integer field, letters and decimal rejected
4	2	Signed float field, letters rejected, but point allowed
8	3	Unsigned integer, letters, point and - sign rejected
16	4	Unsigned float, letters and - sign rejected
32	5	Integer field has minimum limit
64	6	Integer field has maximum limit
128	7	Only letters permitted, numbers rejected
256	8	"password" style field, hide characters typed
512	9	numer of characters entered is limited

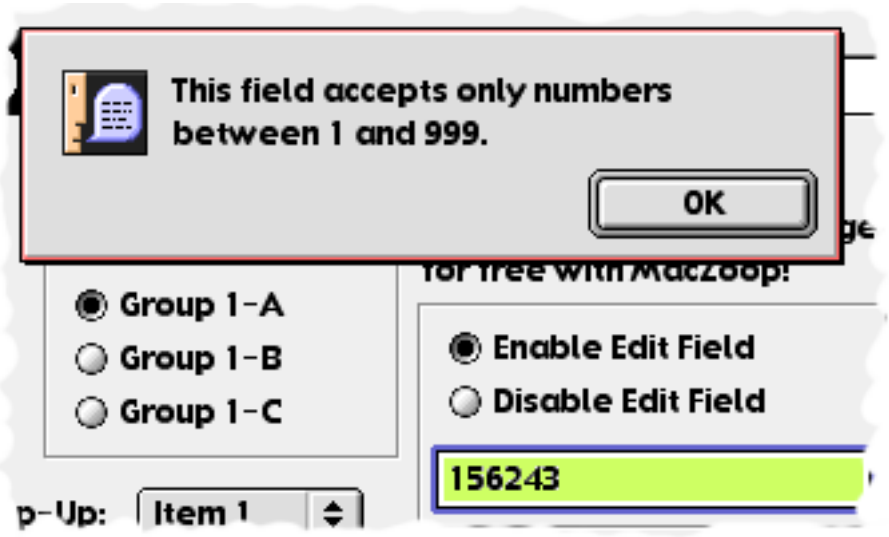
Note that a flag value of 1 (bit 0) is reserved and thus the flags value must never be odd.

Some flags may be combined together. For example, an unsigned integer field with both min and max limits would have bits 3, 5 and 6 set, so the flags value would be 104. Most numeric field entries operate by rejecting the characters that are not permitted as they are typed, beeping instead. This is in keeping with the philosophy of prevention being better than cure. The checking of field value limits however, is done when the user attempts to OK the dialog, in which case the dialog will not close if the value entered is out of range, and an alert will be displayed indicating what the limits are, for the user's benefit. The offending field will be automatically highlighted allowing them to give it another go. When min or max limits are flagged up, the corresponding values can be specified in the edit field's string, following the flags value, separated by commas, e.g.

"1200//104,1000,2000"

means that the field accepts unsigned integer entry only, with min and max limits set, with a min limit of 1000, a max limit of 2000, and an initial value of 1200. Cool, huh? Alternatively, these various limits and values can be set programatically. Integer values can take the full range of a

signed long (32-bit) value. This screen shot shows what the user sees when an entry is not validated when the dialog is closed.

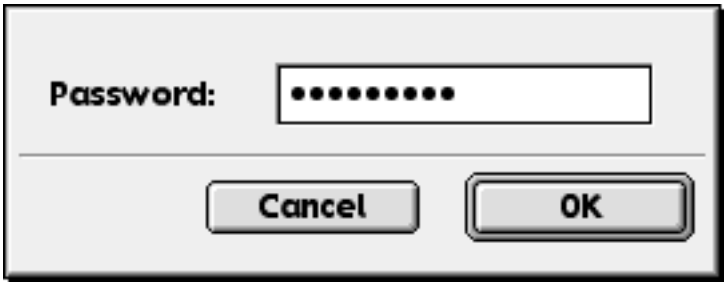


You'll notice that an edit field cannot take a group value. However, edit fields can be enabled and disabled but this must be done for each such item individually. MacZoop ensures that disabling a field changes the input focus to another field and back automatically.

Password Entry

To prevent the casual observer from seeing what a user types when a password is required, it is usual to hide the characters typed into the entry field, replacing them with a suitable glyph. MacZoop provides this functionality as standard. It must be emphasised that this does not mean that the password as typed is in any way encrypted, it merely hides it from the casual observer, and takes steps to ensure that e.g. copy and paste are forbidden as a way to reveal the password. The programmer is expected to manage the storage and handling of the entered password in whatever way is appropriate.

To hide entered text, set the flag with value 256 (bit 8). MacZoop replaces the typed characters with a bullet ('•'). Copying to the clipboard only copies the bullets. Pasting is not permitted. To extract the real text, the programmer uses the GetValueAsText() method that is always used to return entered text from a field. For a password field, MacZoop knows to return the true text. This screen shot shows an example dialog containing a password field.



Other Standard Items and User Items

All standard dialog items- static text, icons, pictures are supported. In addition, they can be enabled and disabled. ZDialog also handles user items. These are non-specific items that the Dialog Manager reserves for custom items in the traditional Mac programming model. However, they are somewhat redundant in MacZoop because the extension items take care of custom functionality. Thus MacZoop turns these over to another very common dialog requirement- the creation of decorative lines and boxes used to group related items together. While this can be overridden for other items, usually the defaults will provide an excellent and simple way to create separator lines and boxes. Simply create a user item for the required lines (1 pixel or less thick) or box (any other size) and they will be drawn in the current appearance style, or a default 3D look for colour dialogs and a 50% grey pattern for black and white ones. When grouping controls into a box, the user item should have an ID number higher than any of the buttons, or hit detection will not work (items are searched in numerical order, lowest first, for hits). Items that overlap the edge of a line or box, with a lower numbered ID, will neatly “knock out” the line at that point, so neat titled boxes are easy to create by simply labelling using Static Text.

Extension Items

MacZoop provides additional dialog user-interface items. These go beyond the standard items to fill in the gaps in the Dialog Manager that programmers often need. For example, to create a list box in a dialog using the Dialog Manager is usually painfully difficult. In MacZoop, it requires nothing more than a little creative work in ResEdit. Extension items are specified in the DITL editor using so-called “magic” strings. In fact, these are not really magic, just a simple way to specify the class of a dialog item, and provide a list of parameters to it. Note that MacZoop lacks a “new by name” feature- you cannot create objects at run-time that were unknown at compile time. Thus while a variety of extension items are provided, if you want to create your own, usually a subclass of ZDialog will be needed in order to make the right object in response to the “magic” string. Magic codes are nothing more than a four-character code that stands for some object type. Mapping this to a real class at run-time needs to be done to extend this mechanism. This gets round the lack of new-by-name at the expense of having to subclass ZDialog.

However, a subclass is not needed for the standard extensions, and if more are added in the future, the base ZDialog will be updated to handle them, so future enhancements will be transparent. The current set of standard extensions are:

- A 1-column list box, with vertical scroll bar
- A 1-column list box with icons and scrollbar
- A scrollable text box, with styled text, optionally editable, and vertically scrollable
- A pop-up menu for picking colours from a 16, 81 or 256 colour palette of your choice
- A progress bar
- A scrollable GWorld item for displaying graphics, previews, etc.

There are also planned to be:

- A digital display
- A clock control

At the time of writing these were not yet implemented.

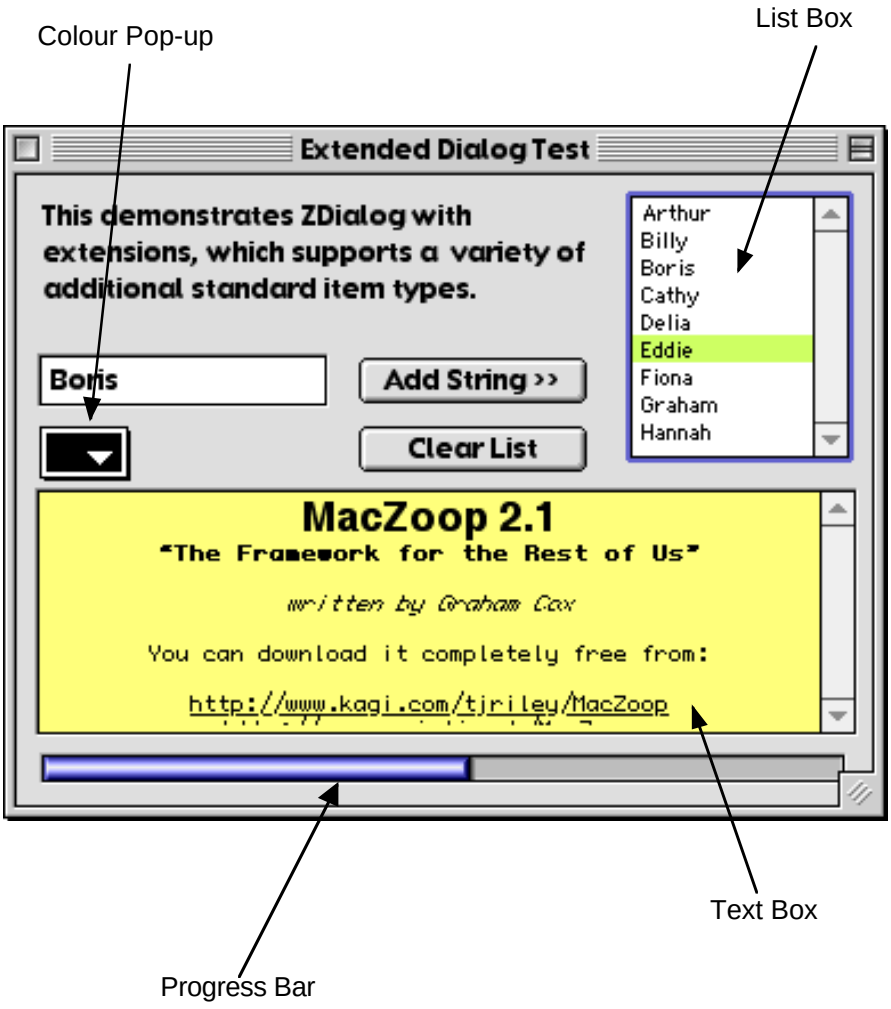
To create such items, you specify them as part of your standard DLOG/DITL resource templates. Initially all extension items start off as static text items (not User Items as you might have expected). This is to allow a string of parameters to be entered as part of the item. To distinguish a normal static text item from an extension item, the special delimiter '$$' is used. The syntactic specification of this "magic string" is:

"$$<class code>[,<param1>[,<param2>[,....etc.]]]"

Up to ten parameters may follow the class code. The class code is a four character code, which is usually intended to be a mnemonic for the type of item. The following table lists the codes for the standard and proposed extension items

Class Code	Item
LIST	list box with strings
TEXT	scrollable text box
ICLB	list box with icons
CPOP	colour pop-up menu picker
PBAR	progress bar
GWRP	GWorld preview
DIGI	digital numeric display
CLOC	clock control

Note that in addition to the above codes, all codes with all lower-case letters are reserved. When

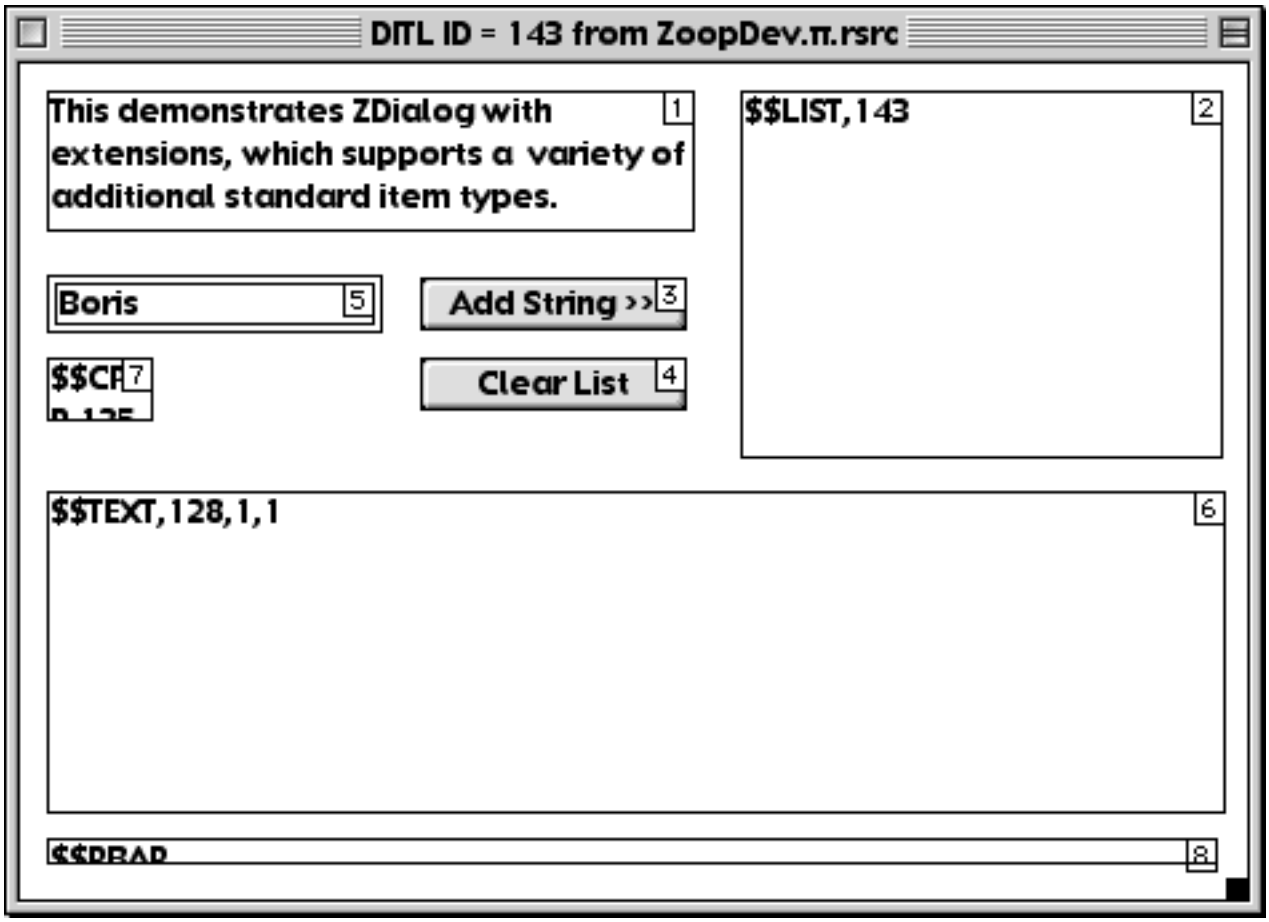


you create you own codes, please ensure they contain at least one upper-case letter. This screen-shot shows some of the standard extension items.

Other items in the dialog are standard buttons, static text and edit fields. It is useful to compare this with the DITL template for the same dialog (shown below), which clearly illustrates how the “magic strings” are used. Some of the strings are obscured because they are wrapped to fit the bounds of the item, but these strings are never visible to the user.

Parameter Lists

The parameters are a way to pass values to the item as it is constructed to further specify its



behaviour or value. Each item type requires different parameters, so we need to list the parameters for each item type. Note that such parameters are always numeric, but they can be interpreted in whatever way makes sense for the object itself- values, flags, resource IDs, etc.

Parameter	*Effect*
List Box Item ‘$$LIST’	
1	optional resource ID of a ‘LIST’ resource (see below)
2	optional resource ID of a ‘STR#’ resource used to fill the list
Text Box item, ‘$$TEXT’	
1	resource ID of a ‘TEXT’ and ‘styl’ resource used as its text
2	flag, 0= not editable, 1=editable
3	justification value, 0=left, 1=centre, 2=right

Progress Bar item, ‘$$PBAR’

1	mode flag, 0=proportional, 1=indeterminate (barber’s pole)
2	max value, if 0, max = 100
3	optional resource ID of background pattern
4	optional resource ID of foreground pattern
5	optional resource ID of striped pattern

Colour pop-up, ‘$$CPOP’

1	resource ID of ‘CLUT’ resource to display

Note that all missing parameters are set to 0, and this should be taken to mean the default value for that parameter should be used. Objects should be designed to be tolerant of missing or badly formed parameters, since this is something that is very easily hacked!

The bounds rect of the item is inherited from the original static text item, and the “enabled” flag that is set by the DITL is passed to the item as the ‘enabled’ data member. Usually, this is used to reject clicks on the item, but the object is free to interpret it as appropriate.

Item Keyboard Focus

Because dialog items are sub-commanders of the dialog, they are permitted to take the keyboard focus if required (see command chain). For instance, the standard edit field has the keyboard focus when it is active (as indicated by a focus border). Typing and menu commands go to the active item first, then the dialog, then the dialog’s boss, etc, all the way up the command chain as usual. Every dialog item has a <canTakeFocus> data member (a boolean) that is set if the items is able to take the focus. This is automatically set for edit field items. Other standard items can’t usually take the focus, but some of the custom types can, for example list boxes and text boxes, if editable. Tabbing in a dialog automatically moves from item to item that can take the focus, inclusing custom items, highlighting them with an appearance-savvy focus ring. Tabbing order is the same as the item number order. Shift-tabbing reverses the order.

Item Resizing

ZDialog allows a general dialog to be resizable where appropriate. The resizing is handled automatically, as per ZWindow. How the items within the dialog are themselves repositioned when this occurs is up to the programmer, but MacZoop makes it easy to define fairly sophisticated behaviour which is then handled automatically. Note that by default MacZoop does nothing special- items will not move nor resize if you don’t program them to do so.

Each item can have a set of sizing behaviour flags passed to it, via the SetAutoSizing() method. The flags are arranged in sets of four, each representing one side of the rectangular item bounds in the order top, left, bottom and right. Each side can move independently of the others, relative to the edges of the window itself. By adding the various flags together, all sorts of behaviour can be specified. The internal arrangement of the flags is a little complex, so MacZoop provides a handy macro for putting together the options value you need.

Each side can:

- stay where it is no matter what the dialog does (the default)
- stay at a fixed distance from the top or left edge of the dialog

- stay at a fixed distance from the bottom or right edge of the dialog
- move a proportional amount to maintain its original relative position

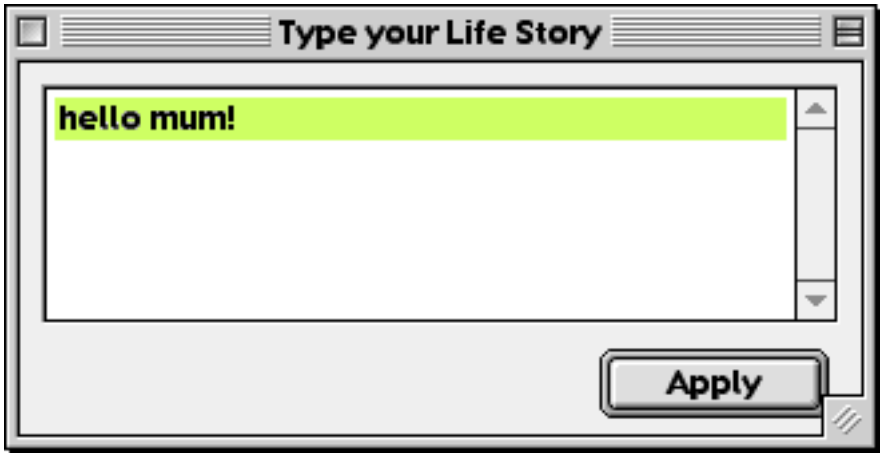
This last can be thought of as a kind of spring, attaching the item's edge to both sides of the dialog, and maintaining the relative distance between the two.

When the dialog is resized, the flags are examined for every item, and then each side of the bounds rect is recalculated accordingly, resulting in a new bounds rect. The items bounds is changed, then the object itself is informed so that it can take any further steps needed to deal with the change. For example, the text box item needs to modify its TextEdit stuff and recalculate the line starts, etc. When designing new dialog items, you need to be aware of the possibility of the item being resized, and deal with it accordingly.

To set up the resize parameter for an item, use the AUTOSIZE macro. This takes four parameters, which are the individual flags for the four sides top, left, bottom and right in that order, and packs them into a SizingOptions value. The flag values are:

```
enum
{
      NONE            = 0,
      FIXEDLEFT       = 1,
      FIXEDTOP        = 1,
      FIXEDRIGHT      = 2,
      FIXEDBOTTOM     = 2,
      PROPORTIONAL    = 3
};
```

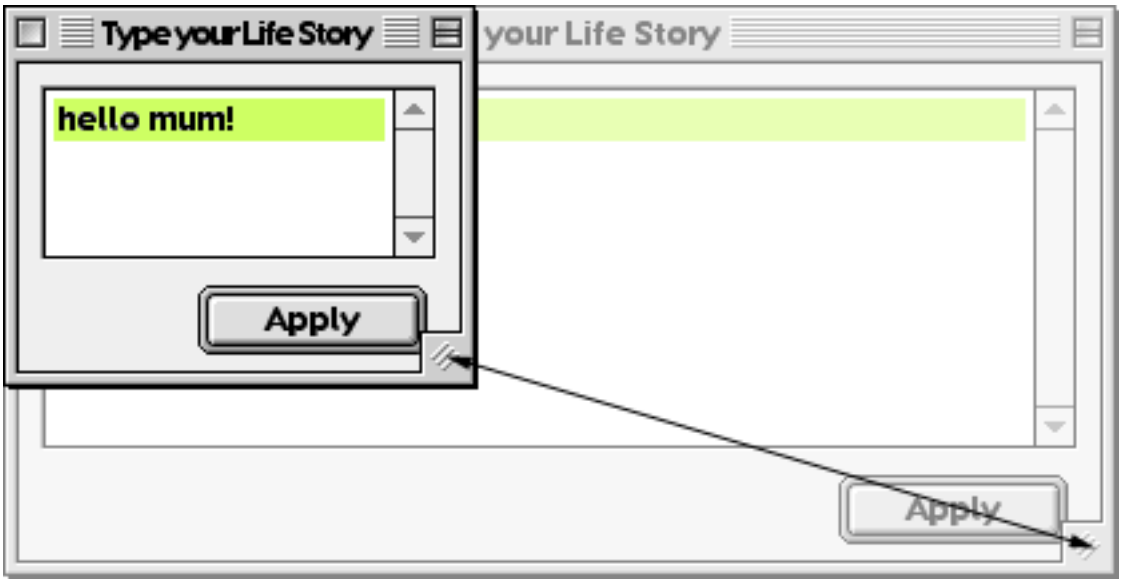
so, let's look at a typical example. We have a dialog thus:



When we resize it, we wish the "Apply" button to stay the same size, but remain at the same distance from the right edge of the dialog box, whereas we want the main text field to expand to fill the available space. To do this, we set the right and bottom edges of the text item to 'FIXEDRIGHT', and the top and left edges to 'FIXEDLEFT'. For the button, we also set the right and bottom edges to 'FIXEDRIGHT', but since we don't want to resize the button, we set the top and left edges to 'FIXEDRIGHT' also. Thus we call:

```
theButton->SetAutoSizing( AUTOSIZE( FIXEDBOTTOM, FIXEDRIGHT, FIXEDBOTTOM, FIXEDRIGHT
));

theTextBox->SetAutoSizing( AUTOSIZE( FIXEDTOP, FIXEDLEFT, FIXEDBOTTOM, FIXEDRIGHT ));
```

Lo and Behold, when the dialog is resized, the items move as we wish:



What would happen if the left edge of the button was set to PROPORTIONAL, instead of FIXEDRIGHT? Try it and see!

Dialogs in Practice

OK, so we know how to do all sorts of tricks with dialogs, and we've been briefly introduced to MacZoop's powerful enhancements. But what is a dialog for? Usually, it's for getting some values from the user before contnuing with another process, or for interacting with the user in some specific way, which affects some other data. In either case, we usually use dialogs as a secondary interface to some other primary interface, such as a document window- the dialog supplies supplementary information which is used in some way on the main data. This implies that we must have an efficient means of getting data out of a dialog box and applying it elsewhere. There are numerous ways to do this, and MacZoop is typically versatile.

Dialogs are commanders- they can respond to commands. This also means that they can (and always do) have a boss object that manages them, and very often this is another window, perhaps the main document. It's also common that a dialog's boss is the application itself, for example a preferences dialog is often arranged like this. Usually, the data entered into a dialog is of interest to the dialog's boss, rather than the dialog itself. While it is possible to use a dialog as a main interface, and for many applcations this makes perfect sense, the dialog itself is not normally the place that the information entered there is actually used or applied.

In other words, we need a good line of communication between a dialog and its boss. The dialog handles the user interaction, the boss does something useful with the information the user supplied. MacZoop provides a number of methods for handling this communication. Which one is most suitable at any one time depends on the exact situation within the application, and commonly all the techniques will be applied at different places within a single app.
The main means of extracting data from a dialog is the GetValue() method, and its cousins GetValueAsText() and GetValueAsFloat(). There is a corresponding SetValue() method for setting data in a dialog to some initial value. Normally, all dialogs in MacZoop are asynchronous.

This means that once a dialog is created and displayed, MacZoop goes back to the main event loop, and interaction wit h the dialog is handled in the normal course of events. This is true even for dialogs which behave modally, locking out interaction with other windows on the screen. While this is consistent, simple and neat, it's also inconvenient in one respect- the place where the dialog was invoked is long since gone when the user finishes their interaction and resumes with the main program.

One way to solve this is to use a so-called "inline" dialogs. This is closer to the normal modal way of handling a dialog, where the dialog is invoked, and when closed, code execution resumes at the same place that the dialog was initiated. This is the simplest way to use a dialog, though not always the most appropriate. MacZoop knows when a dialog is being handled in this manner, and will keep the objects that make up the dialog around long enough after the user closes the dialog for the data within to be extracted. To ensure this, the programmer agrees to accept responsibility for disposing of the dialog when finished with it. Normally, asynchronous dialogs dispose of themselves when closed.

Handling a dialog inline

Inline dialogs should only be considered when you have a situation where the dialog is needed to provide some supplementary information to a command. The command is invoked, perhaps from a menu, and the program responds. For example, lets say we have a command that blurs an image. We call up "Blur..." from the menu, but before we can do the blur, we need to supply an 'amount' value. So a dialog is put up, asking for the amount. The user enters it and hits OK, the value entered is extracted and the blur operation proceeds. The operation is interactive, but only to a limited degree. The user may cancel or proceed, but can't, once the dialog is up, do anything else, such as "Sharpen". A blur "mode" has been entered, for the moment. Since modes are generally bad, this sort of application design should be used only as absolutely necessary.

To use a dialog inline, the general procedure is:

1. Create the dialog
2. Call its RunModal method
3. If it returns TRUE, extract the data using GetValue, etc.
4. Dispose of the dialog
5. Continue processing

Here's some code:

```
void MyCommander::DoInlineBlur()
{
	ZDialog*	dd;
	long		blurFactor;

	FailNIL( dd = OpenSubDialog( kBlurParamsDialogID ));

	if ( dd->RunModal())
	{
		blurFactor = dd->GetValue( kItemFactorEntry );

		DoBlur( blurFactor );
	}

	delete dd;
}
```

Modeless Dialogs

A more user-friendly design for this kind of application might be a modeless dialog. This can live on the desktop among the other windows, and when the user wishes to apply the comand, she brings the dialog to the front, and hits a Blur button. The dialog contains a field for the parameter value, and this can be entered at any time, whether or no the blur is done there and then or not. This is better than the modal dialog, because the user retains the flexibility of what to do and when. The app feels more responsive and easier to use. Because modeless dialogs are just like any other window, when the “Blur” button is hit, we need to be able to respond to that from wherever we are, whatever we are doing, extract the parameter value and execute the command.

The dialog’s boss is ready to receive the message from the dialog that the button was hit at any time. Remember, ZDialogs are Commanders, and all Commanders are also Comrades. Because Commanders automatically receive all comrade messages from objects they manage (are the boss of), we have a ready-made mechanism for solving our dialog problem. When a button in a dialog is clicked, or any other item for that matter, the dialog broadcasts a message to whoever wants to know (and always its boss) announcing this fact. The boss commander can detect the message, extract the item clicked, extract any other data it wants, then proceed with the comand. This is not all that different from the inline case, except that we have done things in two parts. First, we created the dialog. The dialog sticks around and at some later stage, the user maybe interacts with it. In the meantime, the app just goes back to handling events and quietly doing its thing. In the second part, the boss commander (the same object usually that created the dialog) receives a message to say that one of its dialogs was messed with. It extracts the data from the dialog, and then goes off and handles the command.

Here’s some code:

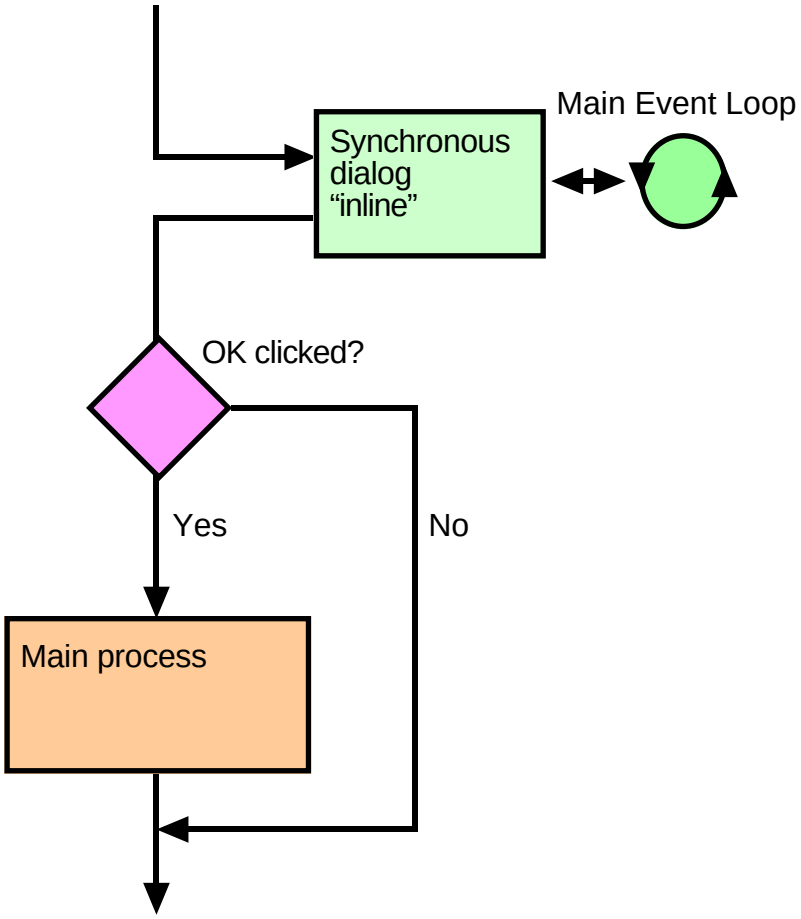
```
fBlurDialog = OpenSubDialog(  kBlurParamsDialogID );
```

and elsewhere, inthe ReceiveMessage method:

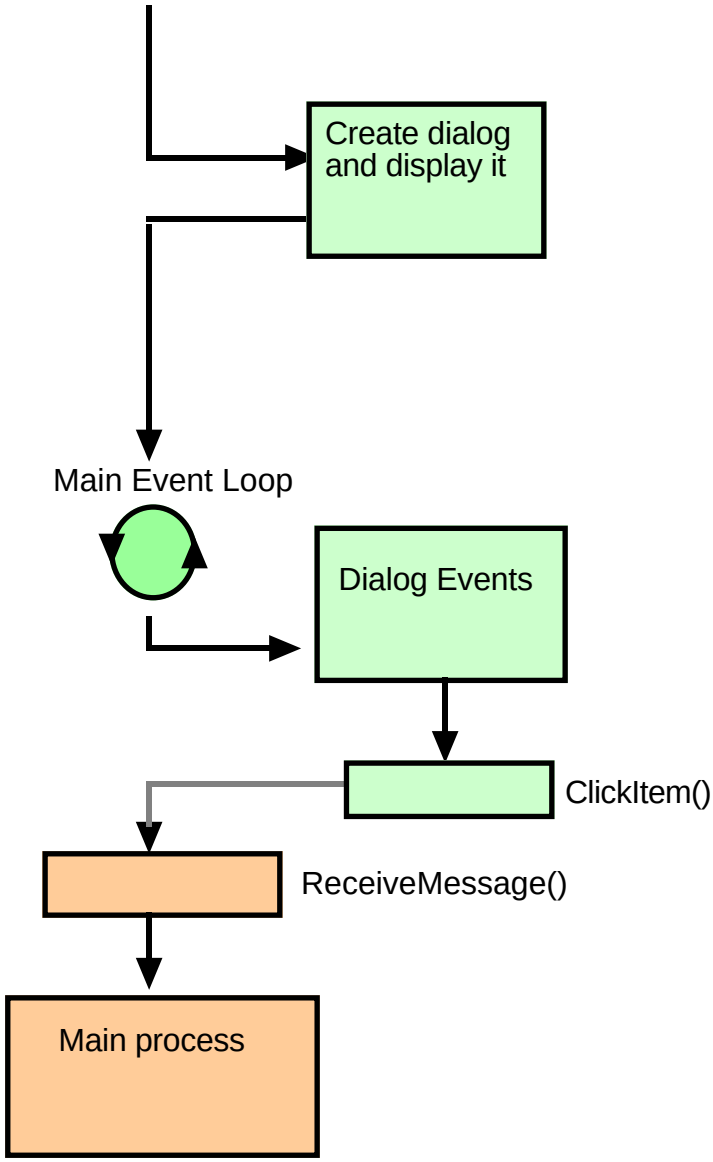
```
void MyCommander::ReceiveMessage( ZComrade* aSender,
                                                  long msg,
                                                  void* msgData )
{
     short dialogID;
     short itemID;
     long      blurAmount;

     if (( aSender == fBlurDialog ) && msg == kMsgDialogItemClicked )
     {
          dialogID = HiWord( *(long*) msgData );
          itemID = LoWord( *(long*) msgData );

          switch ( itemID )
          {
               case kApplyBlurButton:
                    blurAmount = fBlurDialog->GetValue( kItemFactorEntry );
                    DoBlur( blurAmount );
                    break;

               default:
                    // other items
                    break;
          }
     }
}
```

A similar technique can be used for modal dialogs, since although the dialog is automatically closed and deleted when OK or Cancel is hit, before that happens, the boss will receive a kMsgDialogSuccessfullyClosed message. It is fine to call GetValue() to extract data at that time.

These diagrams show the difference between asynchronous and synchronous dialogs. In general, although the asynchronous method looks more complex, it is a preferable method.



Synchronous or "inline" dialog



Asynchronous dialog, e.g. modeless.

Multiple Dialogs, Multiple Bosses

Often there will be more than one dialog associated with a given commander. When handling dialogs in ReceiveMessage(), it is vital to distinguish them. The “sender” parameter is the dialog itself. Also, the dialog ID is passed as part of <msgData>. In the case of kMsgDialogItemClicked, this is the high word of the long whose address is passed in msgData, i.e. dialog ID = HiWord(*(long*) msgData) and for kMsgDialogSuccessfullyClosed, msgData is the address of a long with the dialog ID in both high and low words (this fact may simplify your code for the two cases). Remember, the dialog ID is simply the resource ID of its DLOG resource, that you passed originally when it was created.

More complex to deal with is the case of when a dialog is shared among a set of similar other objects. This is not the same as a global dialog, such as preferences, where the application is the boss. This situation can occur if you have a particular document type but a common dialog for all of the instances of them. An example might be a Find/Replace dialog, shared among a number of text documents. While this could be modal and thus handled the same as before, a modeless design is usually better. In this case, the dialog is the active window, so seeing which document it applies to may be a problem. However, if the programmer wishes to share adialog among a number of documents n this way, some care is needed to manage it correctly. usually, the dialog reference is stored globally, so that all documents can “see” it. Any one of them could create the dialog, but it is assigned to the global instead of a local data member. The document must check the global to see if another created the dialog beforehand before doing the same. The boss of the dialog should usually be the application, rather than a document which may disappear. Upon creation, the document sets itself as a listener of the dialog. ReceiveMessage() works as normal, but all documents will receive the message simultaneously. It is up to the document to determine ifthe command fromthe dialog is directed at it, and one easy way is to compare itself withthe value returned by gWindowManager->GetNthWindow(2); The dialog itself will be window 1, so it’s target is usually the SECOND window. Note that the supplied class ZHexEditor employs this scheme for its Find dialog- browsing that class may be instructive if you are intending to do the same.

Nesting Dialogs

In the same way that a window is often the boss of a dialog, a dialog can open further dialogs as child windows. While excessive use of this can lead to horrible, nasty applications from the user’s point of view, there is nothing to stop you doing it if you so wish. There is one issue to be aware of, and that is, if you have a modal dialog (or moveable modal) and open a child window, it should probably always be modal too. MacZoop won’t stop you if you open a modeless dialog as the child of a modal window, but this will lead to some bizarre behaviour in the application- the modal dialog will not permit you to switch windows, but its child will, leading to a funny state within the app. Needless to say this is highly confusing to the user (though again, not technically wrong- it won’t crash for example) and should be avoided. If you have a moveable modal dialog, its children should be either moveable modal or modal dialogs, and if you have a modal dialog, then any children of this should be modal too. To open a child dialog requires nothing new, simply call the parent dialog’s OpenSubDialog() method.

Dialog Disposal

In all cases where an asynchronous dialog is used, some care may be needed to avoid stale references to the dialog. This is particularly acute with the multiple bosses scenario. If the user closes the dialog’s window, the dialog will be shortly deleted. If any object has a reference to it, that reference will be stale after deletion, and accessing the dialog will crash the program. The solution is to respond to kMsgDialogSuccessfullyClosed by, among other things, setting your dialog references to NULL.

Stack-based Dialogs

Dialogs may be instantiated as stack objects, but ONLY if they are to be run inline. In fact, this is recommended for inline dialogs, since the scoping rules ensure the dialog is deleted at the right time. All other dialog code generally assumes that a dialog is a heap object.

Dialogs in detail - "how to use" guidelines

Creating Dialogs

A dialog must have some commander object as its boss, either a window object of some kind or the application. To create a dialog, call the boss commander's OpenSubDialog() method:

```
ZDialog*  ZCommander::OpenSubDialog( const short dialogID );
```

This method takes the ID of a 'DLOG' resource, and returns the object reference of the created dialog. This method creates the dialog as an underling (child window) of the commander object, and makes the dialog visible and selected. The dialog style, type etc. is established by the resources read.

Of course, you can create dialogs by the more traditional method as used for windows. This may be needed if you are opening the dialog on behalf of another object, or want to do something special, such as keep it invisible for the moment. The correct procedure is:

```
ZDialog*  zd;

FailNIL( zd = new ZDialog( this, myDLOGResourceID ));

try
{
     zd->InitZWindow();
     // make visible here if desired
     zd->Select();
}
catch( OSErr err )
{
     ForgetObject( zd );
     throw err;
}
```

You should recognise this as being identical to the creation procedure for ZWindow. As with ZWindow, the InitZWindow() call is vital. The parameter <this> above is the dialog's boss. While it is usual to pass <this>, opening the dialog from within its boss, you are not required to do so. For global dialogs such as a preferences dialog, this is commonly gApplication.

The InitZWindow() call actually builds the dialog. The construction procedure is:

1. The Macintosh dialog is built from the DLOG/DITL resources by calling the Dialog Manager's GetNewDialog() function. Whether the dialog floats or is modal is determined automatically by detecting the window style requested.

2. The DITL is parsed and a set of ZDialogItem objects is built for each one. These are stored in the <itsItems> list, a local ZObjectArray. How this is handled in detail is discussed below.

3. After all the items have been constructed, the SetUp() method is called. The default method simply selects the first focusable item, and broadcasts the kMsgDialogSetUp message. You can override this method to do further set up if needed.

4. All items are initially disabled, they will be enabled by a subsequent activate event.

5. The window is added to the MacZoop window manager. This knows that modal dialogs go in front of all other windows. Modeless dialogs behave as normal windows.

6. The original position of the dialog on screen is restored (if this feature is enabled).

Now we discuss how the dialog items are constructed. You need to know this if you wish to extend the types of item that your dialog will have. For standard dialogs, or those with only the supplied extension items, this will work whether you know how or not!

The method BuildDialogObjects() is called by InitZWindow() to deal with constructing the items according to the DITL resource. This method walks the real DITL handle in memory by indexing through using GetDialogItem, a Dialog Manager call. The original type of the item is examined, and if a static text item, the string is further parsed to see if an extension item is required.

For each item:

1. The basic Dialog Manager information is requested (ID, data handle and type code)

2. If a static text item, the string is parsed to extract any “magic” code, plus parameters.

3. MakeItemObject() is called. This creates a new ZDialogItem or derived object. If you are extending the range of types, you need to override this method to map your “magic” code to a class name. The returned object is added to the items list.

4. If the item is a control, the title is parsed to extract any group ID. The original title is extracted and passed back to the control, so the user never sees the additional data attached to the string.

5. If the item is an edit field, the string is parsed to see if there are any special flags and other data. If so, the item is set up accordingly.

6. Any colour or font data in an associated ‘ictb’ resource is extracted and passed to the item. This means that the usual mechanism for colouring dialog items works correctly for MacZoop dialogs. Unfortunately, a tool such as Resorcerer is required to set this up, since ResEdit does not do this.

7. The item’s InitItem() is called, with the original set of “magic” parameters passed. The item uses these parameters in whatever way it sees fit, and for the supplied extensions, the meanings are given earlier in this chapter (section xxx, ...).

8. Finally the dialog’s UserInitialise() method is called, which is your opportunity to do any extra per-item initialisation. The default method does nothing.

More on dialog items and how to create your own custom types later.

Note that after construction, there are two dialog item lists in memory. There is the original Dialog Manager one, which never goes away, and the new list of ZDialogItem objects we’ve created. After construction, ZDialog relies on its own list completely, and never referes to the original one. However, this comes into play when we stream a dialog, so we keep it around, even though it seems wasteful. In fact it does not usually occupy a significant amount of space by modern standards, so we can afford it!

Dialog Interaction

Once constructed, the dialog should be displayed, usually by calling its Select() method, as per any window. Selection may be delayed until some later stage if desired.

To make a dialog visible, frontmost and active:

```
myDialog->Select();
```

The user interacts by clicking and typing. It is the items themselves that initially respond to these activities, so let's look at how that's done. First, clicking. Clicks are sent initially to the dialog window, and arrive in the Click() method. ZDialog override's ZWindow's Click() method in order to further resolve the click and pass it to the clicked item. FindItem() is called, which by default uses the Dialog Manager's FindDialogItem() function to determine which, if any, item was clicked. The associated ZDialogItem object is looked up and its Click() method is called. Upon return, the dialog's Click method checks to see if the click resulted in a request to dismiss the dialog, and if so, fakes a click on the dismissal item (usually OK or Cancel in a modal dialog). This is done to provide shortcuts for dismissing a dialog, for example, double-clicking a selection in a list can both pick the item and OK the dialog.

ZDialogItem's Click() method determines the right course of action for the item's type. TrackControl() is called for controls, TextEdit clicks for text fields and so on. If the item is able to take the focus but is not currently focussed, the dialog is called back to change focus to the item first. More about this shortly. For custom items of your own, you will often override Click() to respond to the mouse in your item.

Once the click has been tracked, etc, the dialog's ClickItem() method is called back. This is the main place where click processing takes place.

ClickItem() is responsible for standard click handling such as checkbox flipping and radio button grouping. Also, for modal dialogs, clicks on OK and Cancel are handled here by validating and closing the dialog. Also, vitally, the click message is broadcast from this method. ClickItem() can be overridden in order to do additional item handling if desired, but it is usual to use one of the messages to respond to clicks as discussed above. If you override it, be sure to call the inherited method to get standard click processing for the items.

To programmatically simulate a click on any item, call FakeClick() with the item's ID. This will not correctly simulate a click on e.g. a list- you cannot use this method to select a particular cell. This call acts as though the object was already clicked, to invoke its post-processing. It is particularly effective for buttons, since a short visual feedback of the button going down is provided (8 ticks delay). If you don't want the feedback or delay, you can call ClickItem() directly.

Item Focus and Keyboard Input

Items in a dialog may take the focus if desired. Usually, edit fields can, as well as some other kinds of item. The focussed item is indicated by a highlighted border, if there is more than one item able to take the focus. Focus can be changed at any time by using the tab key (or shift-tab for reverse order). In addution, clicking in a focussable item when it doesn't have the focus will select it before processing the click further.

When an item has the focus, it becomes the start of the command chain, so typing and menu commands, etc go to it first, before being passed to the parent dialog, then the dialog's parent, and all the way back eventually to the application itself. Thus typed characters arrive at the item's Type() method. For standard items, the CheckKey() method is called to determine if the typed character is permitted according to the flags set for the edit field, and if OK, passed to TextEdit.

The tab, return, enter and escape keys are passed immediately to the item's boss (the dialog).

To programmatically change the focus, call SelectItem(), passing the item's ID. If the item can't take the focus, nothing will happen. Alternatively, call SelectNextFocus() or SelectPreviousFocus() to move to the next or previous item in the tabbing order.

Dialog event processing

Other events (not clicks or typing) are handled by ZDialog in a variety of ways. An update event turns up as a call to Draw(), as for any window, and ZDialog responds by iterating the items list and calling the item's Draw() method. Each item draws itself, and in addition draws any focus border and default ring as appropriate.

Activate and Deactivate events result in all items being activated and deactivated (disabled) as needed. This is done by a call to EnableItem() and DisableItem() with a parameter of 0. 0 means all items. The previous state of the item is held in a small stack so that when reactivated, the original state of the dialog is restored. The stack is 32 levels deep, so up to 32 nested activations are permitted. Because a dialog is activated when its Select() method is called, all items are initially disabled when the dialog is created.

The cursor shape is automatically changed as it moves over a dialog- edit fields change it to an i-beam, otherwise an arrow is set. Custom items can alter the cursor shape directly if they wish- the cursor shape is actually set by the item it passes over, not the dialog.

Balloon Help is supported for dialogs using the standard Mac resources for this. This is done by default by calling the item's GetBalloonHelp() method, which by default extracts the help message for an item from the standard resources and sets up the enabled/checked state as needed. You can override this to provide help messages from other sources if you wish.

Command Handling

As ZDialog is a commander, it can handle commands. As it comes, it handles only the standard window commands, such as Close, and Clipboard commands. The close command is honoured for modeless dialogs that have a close box. Modal dialogs do not directly respond to this command (since menus are dimmed when a modal dialog is active), but the default button (OK) effects this command in the same way. ZDialog override's ZWindow's Close() method. First, this calls a dialog specific method, CloseDialog(). This method is provided to allow the dialog a chance to do any closure validation or clean-up, and can be overridden if required. As standard, it simply calls ValidateFields(), which checks any edit fields so programmed for numeric validity. If the dialog is valid, the closure message is broadcast (which you'll recall is the message the dialog's boss is expected to respond to to act upon the OK button) and the window itself is either hidden (inline dialogs only) or deleted. While Close() is called directly as a command for modeless dialogs, it is called by ClickItem() for modal ones. Which actual items are actioned by

this response depends upon the values of the data memebers <defaultItem> and <escapeItem>. These are usually set up to be items 1 (OK) and 2 (Cancel) respectively, so you are expected to set up your dialog this way. However, you can nominate any item numbers for these actions, and the place to do this is in your SetUp() method. It's also possible to supress either button by setting the relevant data member to 0, but this would be unusual.

The clipboard commands Cut, Copy, Paste and Clear are handled directly by overridden ZCommander methods. These use the Dialog Manager functions DialogCut, etc to do their work. This works because the standard ZDialogItem() for an edit field uses the Dialog Manager TextEdit structure for its operations, for maximal compatibility. Other custom items can implement their own clipboard handling as needed, since they startthe command chain when focussed.

The standard Paste command first calls PasteDataIsLegal(), This method is there to ensure that when pasting into an edit field, if any special behaviour flags are set for the field, that the data being pasted does not violate any rules. For example, if an integer numeric field is targetted, and the data on the clipboard contains letters, you will not be allowed to paste to the field. Illegal data on the clipboard results in a beep. If the field is a password type field, all pasting is forbidden.

Inline Dialogs

To use a dialog inline, you create the dialog as normal, or for inline dialogs only, as a stack object. Then you may call the RunModal() method to interact with the dialog. For your convenience, RunModal() will also call InitZWindow() and Select() if you haven't already done so, cutting down on the code you need to write for each dialog. RunModal() returns a boolean. It is TRUE if the default button dismissed the dialog, FALSE otherwise. Normally, you should permit the user to cancel, and only perform the subsequent processing if RunModal() returns TRUE. Note that RunModal() calls the application's main event loop method Process1Event(), though it implements its own loop. This means that timers, etc. still get time while a modal dialog is active. You need to be aware of this in your app design, since this is different to the native Mac implementation of modal dialogs, where usually all main event processing is suspended. The MacZoop way has many advantages- for example if the dialog is moveable, windows behind it get repainted automatically. However, with timers potentially still running, some processing activity may continue to occur when you didn't want it to. It;s up to you to design your app accordingly, or else detect the fact that a modal dialog is active and act accordingly.

To detect a modal dialog, call gWindowManager->GetTopWindow() to obtain the active window, then pass it to the window manager's IsDialog() method. This returns TRUE if the window is a modal dialog, FALSE otherwise. (In spite of its name, modeless dialogs retrun FALSE from this function). You can detect any dialog type by checking the window using a dynamic cast.

Item Input and Output

ZDialog has a single polymorphous method for setting an item's value- SetValue(). This method allows you to set any individual item's state, value or text using one call. You can pass in shorts, longs, floats, doubles and Str255 variables. The item will respond as expected (although passing a string to a control has no effect). Note that using SetValue() to switch on a radio button that is part of a group will turn on that button, but will not turn off its group members. To do that, you should call FakeClick() with the ID number of the button that should be ON.
Extracting data from a dialog item requires the inverse function to SetValue(), GetValue(). This

returns the value of the item as a long. You can also obtain it as a string using GetValueAsText() and as a float using GetValueAsFloat(). It is assumed you know what type of data you want! Any necessary conversions are done.

Each of these methods looks up the dialog item itself and simply passes on to the item's equivalent method, named the same way. Custom Items can override these methods to provide their data in an appropriate way.

To set the title of a control, use the SetItemTitle() method.

To get the ID number of the radio button that is on in any particular group, call GetSelectedItemInGroup(), passing the group ID.

Miscellaneous

To enable or disable an item, call EnableItem() and DisableItem() respectively, passing the item number. Pass 0 to these methods to enable or disable every item in the dialog. This is done automatically when the dialog is activated and deactivated. To enable or disable all items in a particular group, call these methods passing the negative of the group ID.

Items may be hidden or shown using the HideItem() and ShowItem() methods.

Set an item's sizing behaviour for a resizable dialog using the SetItemSizing() method. The AUTOSIZE macro is convenient for packing the flags into the required SizingOptions format. This is discussed in more detail in the main text above.

Extensible Dialogs

Sometimes, it can make for a nice interface to be able to extend a dialog at run-time, while it is displayed. An example might be a multi-panel preferences dialog. MacZoop supports this. he set up for such a dialog is to have a DLOG/DITL which contains the dialog's BASE items, which are the items common to all the varieties of interface in a multi-part dialog. This usually includes the OK and Cancel button, and whetever is used to select the various panels. Each subsequent panel is described using further DITL resources. The items in these further DITLs are added tothe dialog, and positioned according to the DITLMethod parameter, which is documented in Inside Macintosh.

To append a DITL to a MacZoop dialog, use the AppendItemsToDialog() method. This calls the Mac toolbox AppendDITL function and makes sure all the correct objects and other set-up is done correctly.
To remove a set of items added in this way, call the RemoveAppendedItems() method.

For a multi-panel interface, you normally have a control or other selector, and to switch panels, you should remove any existing one, then add the next one in response to a choice in the selector device. ZDialog keeps track of the number of items in the original dialog. You can get this number by calling the GetBaseItemCount() method. RemoveAppendedItems() removes all items except those covered by the base item count. Thus it is possible to call AppendItemsToDialog() more than once to add items, and make a single RemoveAppendedItems() to remove all of those added so far.

The standard Dialog Item- ZDialogItem

All of the standard Macintosh dialog items are embodied in a single class, the base ZDialogItem. You can subclass this for your own kinds of custom item. ZDialogItem is a commander, and the dialog itself is always its boss.

Properties of ZDialogItem

Every item has a number of common properties. These are its type and “magic” type, state variables such as enabled, visible, focused, hilited, a bounds rectangle, a font specification, a colour and various storage handles. Because items have font and colour info associated with them, there is no need for the complex and skanky ‘ictb’ stuff of the Dialog Manager, MacZoop is much more straightforward. The ’ictb’ resource is used to initialise these values, then takes no further part. Thus changing the colour or font of an item is very easy. To chengethe text drawingparameters, call SetFontInfo() with the font, size and style you desire. To change colour, call SetForeColour() and SetBackColour().

The type of an item is the same as the Mac Dialog Manager type, as detailed in Inside Macintosh. This value is made up from a variety of fields and flags, and is very important.The ‘enabled’ flag is stripped off this and moved to the “enabled” data member. The “magic” type is a more readable type identifier, consisting of a four character code. For custom items, this is the same as the original “magic” type specified in the resource. For standard items it is set thus:

Item	*Magic type code*
Static Text	‘stat’
Editable Text	‘edit’
Pushbutton	‘butt’
Checkbox	‘chek’
Radio button	‘radi’
Other control	‘cntl’
Icon	‘icon’
Picture	‘pict’
User item	‘user’

To query the type of an item, call its GetXType() method. To get the original type of the item, call its GetType() method. Note that for custom items, the value obtained by GetType() may not be what you expect, since this can be manipulated to take advantages of certain behaviours of the standard ZDialogItem class. So beware!

Every item has a bounds rectangle. Normally, the bounds can be considered to be the area outside which no drawing will take place, but this is not completely strictly adhered to. For example, an edit field draws its outline rectangle 3 pixels outside this rectangle, and if 3D effects are available, occupies even more space. The focus ring also takes up space well outside this rectangle. The lesson is to leave enough space around every item to allow for this.

Every item has a field called <macItemHandle>, which is usually set to the handle returned by the original GetDialogItem() for that item. This handle should not usually be altered or disposed. For custom items, the handle is used in a variety of ways, or may be NULL. There is another handle (a TEHandle) in every item called <pwMirror>. This is usually NULL, but is used for

edit fields that store passwords for the storage of the real text. Custom items may reuse this handle as needed. Note that standard edit fields do not store their own TextEdit records, but intead use the standard DialogManager TextEdit record for this.

It is not especially worthwhile going through every method of ZDialogItem() here, since by and large they do what you would expect. Some require a little explanation though, and should be borne in mind for creating custom items.

Focus

If an item is able to take the focus, the <canTakeFocus> flag should be set. Edit fields set this automatically, other items do not. When the user tabs in the dialog, ZDialog looks ahead for an item with this flag set, then for the item that already has the focus, calls its BecomeHandler() method with FALSE. The item responds by clearing the <focused> flag, and removing the focus ring, and any highlighting. Then the newly focused item's BecomeHandler() is called with TRUE, and the item responds by setting the <focused> flag, and drawing the focus ring and any highlighting, etc. The dialog stores the current focus object and this can be obtained using the dialog's GetHandler() method.

Custom items should generally NOT override BecomeHandler(), but instead override the methods it calls, such as DoHilightSelection() in order to do the right thing when the focus changes.

Default Items

Default Items, such as the OK button in a modal dialog, have a "default outline" draw around them. To indicat ethat this is the case, the <isDefault> flag is set. Normally, this is done by the dialog. You should not directly set or clear this flag. When the item is updated, if this flag is set, the DrawDefaultOutline() method is called, which draws the traditional 3-pixel round rect aroundthe item. If Appearance is on, this is automatically converted to the correct style.

Creating Custom Items

The full power of ZDalogItem really comes to the fore when you wish to make a new user-interface element. MacZoop comes with some custom items already, but you may need some particular widget that isn't available.

To make a custom item, you subclass ZDialogItem(). You then override those methods as needed to implements its appearance and behaviour (look and feel). Many of the standard methods in ZDialogItem can be used to help you and cut down on the amount of coding needed. For example drawing the focus ring or default outline will almost always be handled by the base class.

For custom items of your own, you need to also subclass ZDialog (or else modify the original) in order to create your object type when you own magic code is encountered. This is done in MakeItemObejct()- you simply respond to the code you declare by making the right object and returning it. The dialog does the rest.

You shoud draw you item in an override to DrawItem() (not Draw()), and respond to the mouse with an override to Click(). Remember to constrain your drawing to within the <bounds> rect, not including any decorative border. Note that ZDialog clips to a rectangle that is 4 pixels larger than the border (8 pixels larger for edit text fields) before it calls your Draw() method. You

cannot draw outside this area at all, including focus rings and so forth.

If your item is a control (a real Mac control that is), you can harness some of the standard behaviour such as dimming and tracking from the standard ZDialogItem by a) storing the ControlHandle in <macItemHandle>, and b) setting the <iType> field to ctrlItem. It's also wise to set the <iType> flag kCustomDialogItemType for all custom items, so that future changes to the base item can be kept compatible. Note that if you manipulate iType to fake a real dialog object, you are responsible for maintaining the illusion throughout, and making sure that e.g. <macItemHandle> is valid! If you don't, you run the risk of having a serious bug in your app, with unpredictable consequences.

Custom Items may use <pwMirror> as they see fit, and are responsible for disposing it when deleted.

Custom Items must be prepared to be resized. SetBounds() can be overridden (or perhaps ParentResized()), in order to take additional steps as needed when the item is resized. There is no easy way for an item to prevent itself being resized, nor is this desirable, so be prepared!

Utility Methods

ZDialogItem provides some methods that all items can use:

ValidItem() removes the item's bounds rect from the update region of the dialog. This is useful to prevent "double-drawing". If you have already drawn the item, you can call this so that it is not upadted again.

InvalItem() is the opposite- it adds the items bounds to the dialog's update region. This is the best way to redraw an item if its state changes.

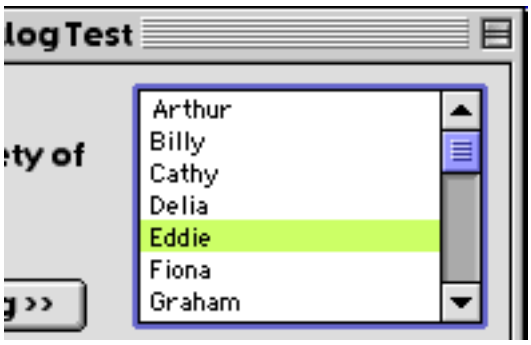
FocusBoss() makes sure that the dialog containing the items is set up as the current port. You may need this if redrawing the item in response to a command, etc, where the current port may not be correctly set. It's also a good idea to use ZGrafState stack objects to record the port's state after setting it, andbefore calling PrepareForDrawing(), which applies the item's font and colour specs to the current port. Careful use of these calls in conjunction withZGrafState is needed to prevent inadvertent cross-coupling of item's styles.

Ready-made custom items

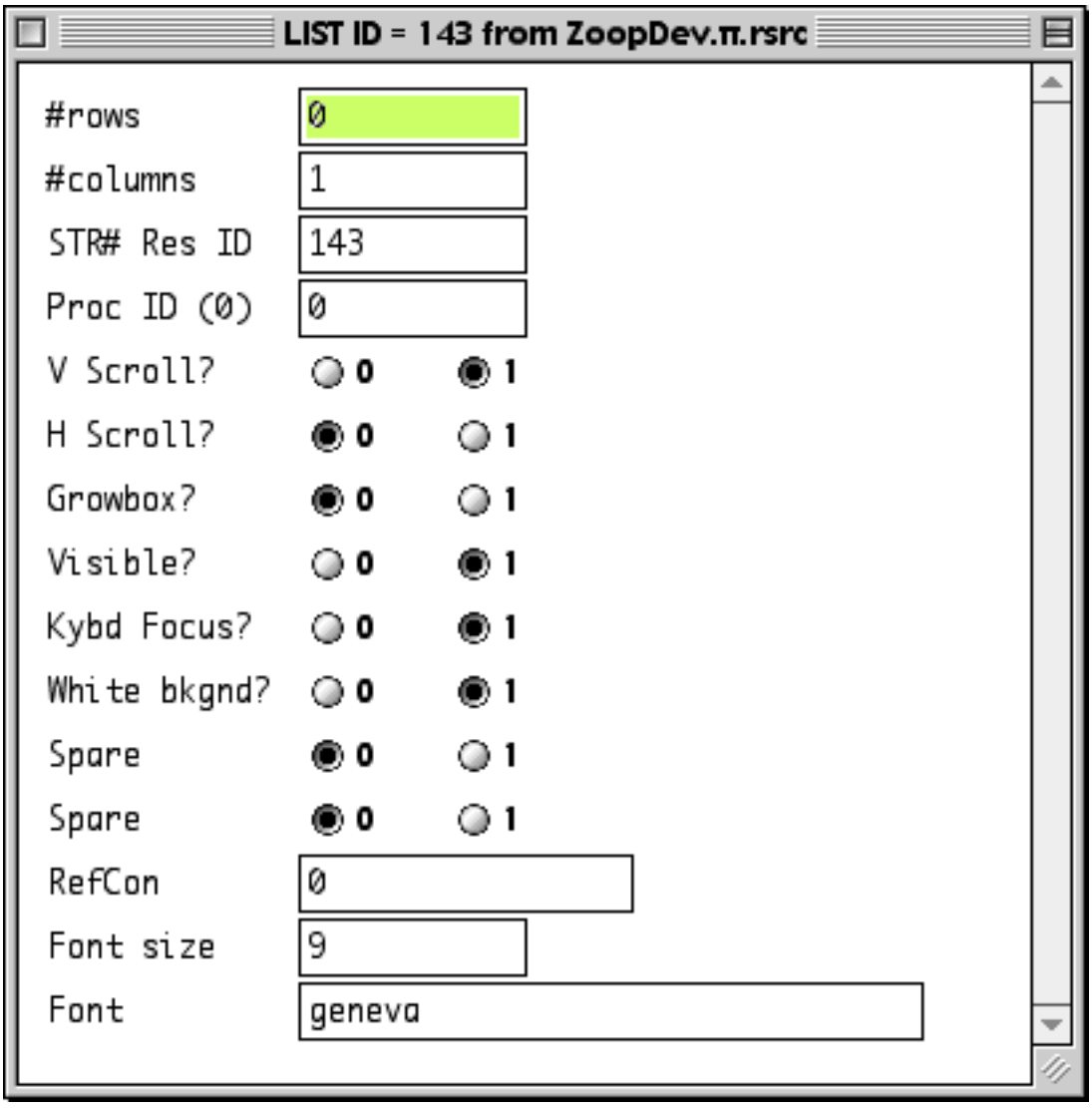
List Boxes

List boxes are dialog items that also inherit from MList to implement scrollable lists of items. The class is ZListDialogItem. Typically, lists in dialogs have one vertical column, though there is no reason not to have multiple columns, or horizontal lists if you wish. MacZoop is biased to providing a 1-column vertical list though, so others may take a little extra work (though not much).

Here’s what a list in a dialog looks like:



The magic type code for a list box is ‘LIST’, and an additional resource can optionally be used to specify the various parameters of the list. This resource has the type ‘LIST’ and the ResEdit template looks like this:



This resource is read by the ZListDialogItem as part of its initialisation, and used to set up the List Manager data stuctures needed. All of this happens without the programmer needing to know much about what's going on, but needless to say all of the tedious steps needed to set up a list in a dialog box are handled for you. The 'LIST' resource is optional. You don't need one if your list has one column, uses the standard LDEF and displays strings using the default dialog font. However, anything else requires one, but as you can see it's pretty simple. You set the number of rows and columns, the ID of a STR# resource used to fill it initially (or 0 if you don't want this done), the Proc ID, usually 0, of the LDEF (jn fact MacZoop can handle custom LDEFs rather more neatly than by using a precompiled LDEF proc, so its usual to set the to 0 even if you have custom lists), the flags for scrollbars, growbox, whether initially visible and whether it can take the keyboard focus. The White Bkgnd flag is a convenient way to specify the common requirement of a list with a white background. If unset, the default background will be the same as the dialog itself. You can also use an 'ictb' resource to set this to any colour. The refCon should be 0 and the font and size can be whatever you want.

ZlistDialogItem knows which LIST resource to read because you pass its ID as a parameter in the magic string:

$$LIST,143

specifies LIST resource 143. If you don't want to use a LIST resource, you can still specify a STR# resource with which to initially fill the list. This is passed in the second parameter:

$$LIST,0,150

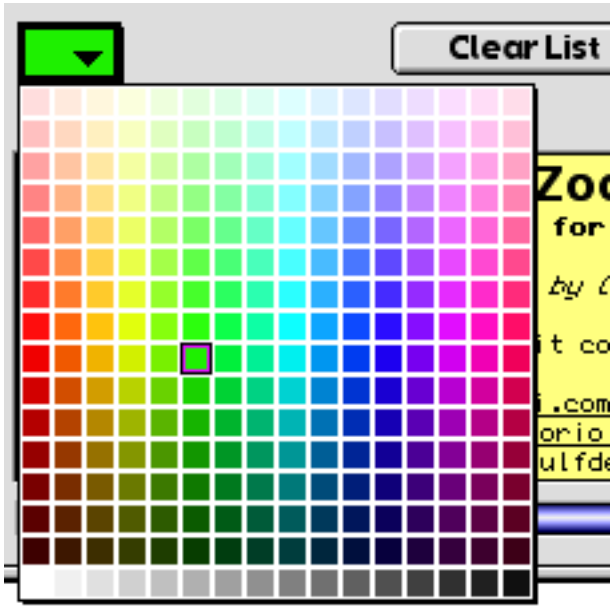
requests a 1-column list with the default font, filled with strings from STR# resource ID 150.

ZListDialogItem inherits from both ZDialogItem and MList, so the API is as per both of those. It does not add any new methods of its own, so there is nothing special about accessing it. If you know how MList is used (e.g. as part of ZMListWindow) then the same API works here.

One feature of ZListDialogItem is that it will tweak its own bounds rect so that an exact number of cells is accommodated, with no partial cells visible. This may cause the bounds to become smaller than set in the dialog template, but never larger. This is a handy feature to help the finished result look more professional, again with no effort on the part of the programmer.

Colour Pop-up menu

The colour pop-up menu is a useful element for very quickly picking a colour from a small fixed set. Currently, ZCPopDialogItem supports colour sets of 16, 81 or 256 colours (4x4, 9x9 and 16x16 arrays). The colours themselves are supplied in a standard 'CLUT' resource, and a custom MDEF is used to display them. Here's what it looks like:



The magic string is ‘$$CPOP’, and the only parameter is the ID of the CLUT resource to use. If you don’t supply this, the system 256 colour table is used. The bounds rect for the item should usually be set to 40 pixels wide by 24 high for the “standard” MacZoop appearance.

The API for the item is simple. To obtain the selected colour, call GetChosenColour(). This returns the selected RGB value. To programatically set the colour, call SetColour(). This accepts an RGB value, and will find the colour in the current CLUT the most closely matches it, and sets this as the current colour. Note that there may be a considerable difference in colour if the palette is small or does not have a wide range of colours.

The CPOP dialog item is, by default, able to be torn off. All that happens is that when the user drags the menu off, the item calls ZApplication’s ProcessMenuTearOff() method. By default this does nothing. Your app may override this to do something special if required. The demo application shipped with MacZoop shows one way to handle this.

Progress Bar Item

The progress bar item is useful to display the progress of a lengthy operation. It is implemented by the ZPBarDialogItem, a subclass of ZDialogItem. This item can be used in any dialog. In addition, there is a complete progress dialog, which is useful for monitoring processes that have no direct dialog interface of their own. Using the complete disloag is covered at the end of this chapter.

The magic string for the progress bar item is ‘$$PBAR’, and takes up to 5 parameters. It looks like this:

If appearance is available and active, the appearance set for progress bars is used. If not, then

the appearance of the bar is established by drawing with a number of colour patterns. You can specify your own patterns for a wide variety of appearances, but by default MacZoop supplies patterns which mimic the appearance of the original progress bar used by the Finder in Mac OS version 7.

ZPBarDialogItem acts much like a control (in fact under appearance it is one). You set a maximum value and a minimum value, then a call to SetValue() draws a bar at the relevant proportion between the two. Incrementing SetValue() from minimum to maximum causes the bar to fill up.

In addition to this behaviour, known as proportional mode, is an additional behaviour called indeterminate mode. This implements the class “barber’s pole” style of progress bar for when you want to give the user feedback that a process will take some time, but don’t know how long (the max value information is missing). The bar looks like this in this mode:

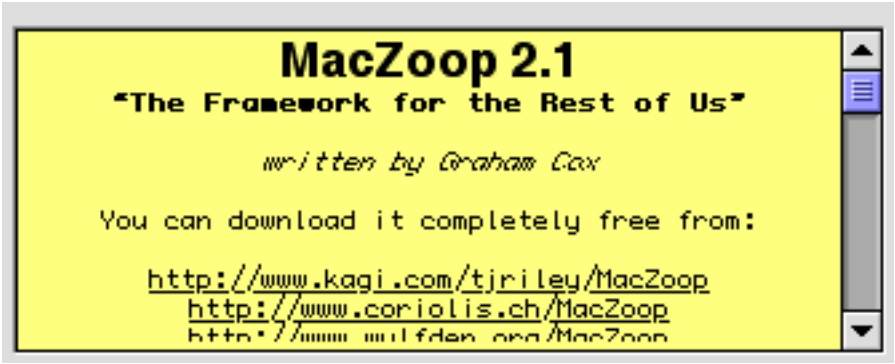
Calling SetValue() periodically will cause the animation of the stripe to occur.

You can switch between one mode and the other by calling SetDisplayMode(). The mode can be initially set using parameter 1 of the magic string. 0 = proportional, 1 = indeterminate. While in indeterminate mode, the value actually passed to SetValue() is ignored, so you can switch from mode to mode on the fly, and the proportional bar will not be updated with an erroneous value.

Scrolling Text Box

A scrolling text box containing styled text, optionally editable, is available as a dialog item in MacZoop. This is implemented by the class ZTextDialogItem, a subclass of ZDialogItem. This uses TextEdit to provide a place to display and edit text that is more extensive than a standard edit field. It can handle up to 32K of text, whereas a standard fiekld can only handle up to 255 characters. In addition, the text is styled and can be justified to left, right or centre as required. The item responds to menu commands relating to text automatically, just as ZTextWindow does.

The text box looks like this:



The magic string for atext box item is ‘$$TEXT’, and it takes up to 3 parameters. The first is the resource ID of a ‘TEXT’ and ‘styl’ resource for the default text. If 0 or missing, the item starts off blank. The second parameter indicates if the user is allowed to edit the text. If 0 or missing,

the answer is no.If 1, the text is editable. The last parameter is the justification parameter. 0 or missing indicates left, a value of 1 means centered text, and 2 means right justified.

By and large, this item will simply work and that's all there is to it. If your app has the standard Font menu and style commands, this item will respond to them when it has the focus.

To programmatically obtain the text from the item, call its GetTexr() method, and to set the text, call its SetText() method. SetText() replaces all current text. SetValue() is also implemented to replace the text with any string as per other dialog items. Note that passing numeric values through SetValue() is unsupported by this item. You can manipulateth etext via TextEdit- to obtain the TEHandle, all the GetTextEditHandle() method.

Resizing this item will reflow the text.

Icon List box

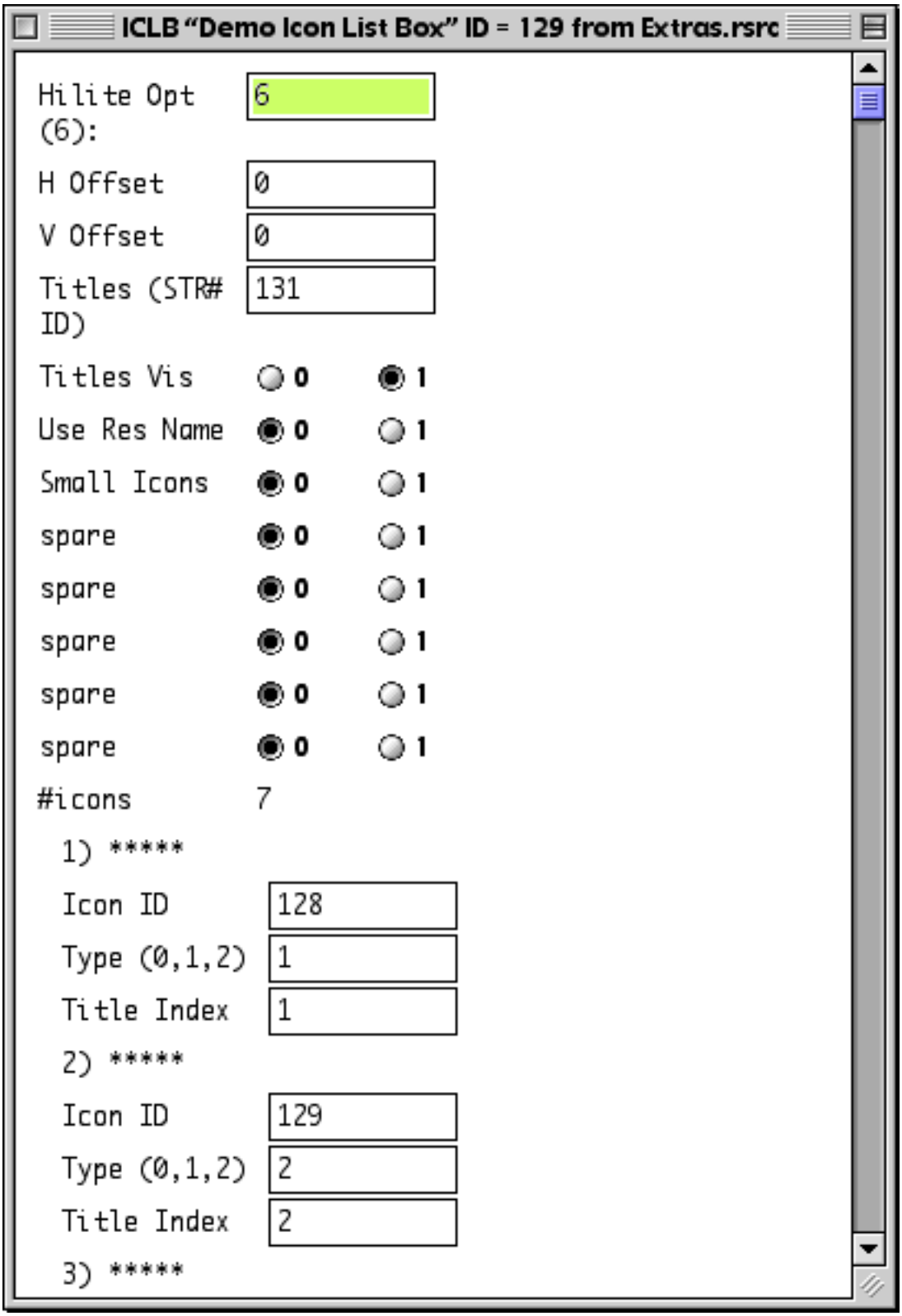
The icon list box is a derivative of ZListDialogItem that uses a custom LDEF to display labelled icons instead of strings. This particular widget is great for implementing the selector device of a multi-part dialog, but has many other uses. here's an example:

The magic string for this item is '$$ICLB', and the class is ZIconListDialogItem, which is a subclass of ZListDialogItem. This class takes advantage of the built-in ability of MList to allow an custom LDEF to be used directly within the code, without all that tedious mucking about compiling and debugging a true 'LDEF' resource. With a few overrides, it does all that is required to handle icons.

Two resources are required. One is a LIST resource, as for the plain list box. This specifies the basic set up of the list, such as the number of columns and rows, etc. The other is an ICLB re-

source, and this sets out the icons to use, what their titles are, etc. Both resources must have the same ID, and are specified by the single parameter in the ICLB magic string.

The ResEdit template for the ICLB resource look like this:



This shows just the first two icon specs, since this is a list-type resource and can have many entries.

The fixed header consists of:

Hilite Option. This dictates how the hilited icon will be displayed. This is a set of flags, and different elements can be combined to give a variety of effects.

Flag Bit	*Value*	*Meaning*
0	1	Draw a bold box around the icon
1	2	Darken the icon
2	4	Invert the title rectangle
3	8	Invert the whole cell
4	16	Highlight the whole cell in the system highlight colour

Usually, the setting of 6 (invert title and darken icon) is the best to use.

HOffset and **VOffset** are used to determine where in the cell the icon is drawn. Thsee should be set to 0.

Titles is the resource ID of a STR# resource which lists the actual strings used for the icon titles.

Titles Vis sets whether to display titles or not. The other flags are, at the time of writing, not implemented and should be set to 0 at the moment. Note that if using colour icons of type ‘cicn’, icons of any size can be used, but the cell size is calculated assuming that the icon is 32x32. Other sizes will therefore work, but may not look good.

Each icon is then specified.

Icon ID is the resource ID of the icon to use. All types of icons- ICON, icn# and cicn are supported.

Type indicates what resource type to go and look for. Type 0 is the basic ‘ICON’ resource, Type 1 means use a ‘cicn’, and Type 2 is a ‘icn#’ family of icons. Be sure to specify the right type or the icon will not show up.

Title Index is the index into the STR# resource used for the titles, to get the string for the title of this icon.

This is all that’s needed to set up the list. The length of the list is calculated from the number of icons- there is no need for the rows# in the LIST resource to match. Font and Size from the LIST resource is honoured- geneva 9 is a good choice here.
List manipulation is done using the stndarad MList API. In addition, ZIconListBox supplies one additional method that may be of interest- AppendIcon90 allows you to extend the list of icons programmatically. Tjis method requires that you pass a structure of type IconInfo(), which isdefined in the header for the class. Filling it in correctly is your responsibility, as is loading the correct icon, etc.

ZScrollerDialogItem and ZGWorldDialogItem

These items are analogous to the classes ZScroller and ZGWorldWindow respectively. They provide a) a generic way to create scrolling content in a dialog item and b) a scrollable GWorld "preview" in a dialog item. Here's what it looks like:

The magic string for the plain scroller is '$$SCRL', and for the GWorld version, '$$GWRP'. The scroller works in a similar fashion to ZScroller, in that there is a rectangle which defines the scrollable area. Since ZDialogItem defines 'bounds' to be the boundary rectangle of the item, we have to use a different term for the scrollable area, so unlike ZScroller, this is called the 'scrollBounds'. The content of the scrollBounds is drawn with the overridable method DrawContent(), as for ZScroller. The scrollable area can be set using the SetScrollRect() method. The parameters that may be passed to this item in the magic string are:

param 1- feature flags

bit 0 = has a vertical scrollbar
bit 1 = has a horizontal scrollbar
bit 2 = has a growbox

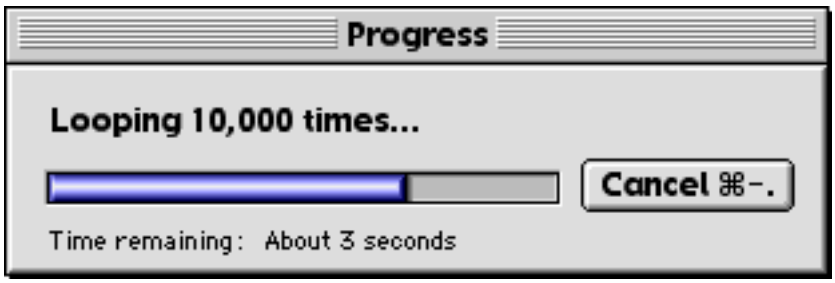
param 2- initial width of scrollable area, in pixels
param 3 - initial height of scrollable area, in pixels

ZGWorldDialogItem subclasses ZScrollerDialogItem to provide a "preview" type dialog item. It uses ZGWorld to provide an offscreen buffer for an image. This can be preset from a PICT resource, or using the parameter list of the magic string. Its parameters are:

param 1- feature flags, as above
param 2 - the ID of a PICT resorce to preload, or 0 if none
param 3 - the width of the GWorld, if no picture
param 4 - the height of the GWorld, if no picture
param 5 - the depth of the GWorld, if no picture. Can be 1, 2, 4, 8, 16 or 32. Passing 0 makes the depth equal to the main screen.
param 6 - the ID of the CLUT resource to use for the GWorld's colour table. Passing 0 will establish the system colour table appropriate to the image depth.

Using the GetGWorld() method, you can obtain the ZGWorld object and make run-time alterations using it as you need.

ZProgress- a complete progress dialog



This class is a ready-made dialog that contains a ZPBarDialogItem and can be used to monitor any lengthy process. As well as visual feedback of the progress, it automatically calculates the time remaining and provides the standard command-period abort command. You can programmatically set the message displayed too. You can also set a threshold time before the dialog is shown. This is handy if your routine takes a variable amount of time, and you don't want the dialog to bother the user if it's less than, say, 3 seconds.

ZProgress is designed to be very easy to add to your code, since as a programmer you don't want to have too much to do to support this. To this end, ZProgress is intended to be used as a stack object, unlike most other dialogs. You can also use it as a heap object if you want to.

By using ZProgress to monitor an otherwise blocking function, you will automatically give up a little time to the rest of the system, without needing threads or other piecewise breakdown of your function. Every time you update the progress bar, event handling and background apps get time.

When you create ZProgress, you pass all the parameters needed to set it up- the dialog ID, the max value, the type of progress box you want (Cancel or Stop are the current options) and the progress mode, proportional or indeterminate. As with all dialogs, the boss must be specified- this can be any convenient ZCommander object.

Here's a complete example of the usual way to use it:

```
void      MyLengthyProcedure()
{
     ZProgress  pd( this, kStdProgressResID, kLoopMax, kCancelType,
     kProportionalProgress );

     long i;

     // begin the long loop

     for ( i = 0; i < kLoopMax; i++ )
     {
          MyLoopProcess( i );
          pd.InformProgress( i );
     }
}
```

That's it, in its most basic terms. You create the dialog (here as a stack object) passing the loop max and various option parameters, then call the InformProgress() method as the loop iterates. By making this one call, you update the progress bar as needed, compute the remaining time, and

also give time automatically to the rest of the app and other running processes. You also get automatic handling of the Cancel button (hitting Cancel will cause ZProgress to throw a silent exception, which will abort the process and delete the dialog). This example assumes that you know in advance the loop iteration maximum, and it codes it as the constant kLoopMax. Of course, this could be a variable- for example when parsing a file, you could set up the progress with the length of the file, and pass the number of bytes read so far as you parse it. Maximum length that you can pass as <max> is the maximum signed long.

This example does not set up a delay or set a message. To do that, you use the SetDelay() and SetMessage() methods respectively. The delay is useful if the length of the operation varies. SetMessage() can be used at any time to display a veriety of different messages as the loop progresses.

The time estimate is based on the amount of time needed to process the amount done so far, and projecting this accotding to the amount left to do. This works very well if the process proceeds at a fairly constant rate. If not, the time estimate may not be accurate- although the estimate will be continually re-evaluated, upward estimates are not posted unless the time increases beyond the previous estimate by more than 20 seconds. Also, the estimate is deferred until at least 3% of the operation has been completed, since smaller amounts lead to less accurate estimates. The time estimate is suppressed if the “barber’s pole” mode is selected.

To use ZProgress, make sure you add the progress resources to your project- either by copying them into your own resource file, or simply by including the ZProgress.rsrc file as part of your project. The standard dialog has an ID of 133, as given by the constant kStdProgressResID. You can also use a custom dialog if needed, but the standard item ID numbers must remain as for the standard ZProgress resources. You can use all the usual ZDialog API to access any additional items in your custom dialog.

How MacZoop's comrades communicate

In any object oriented software, communication between objects is important. Usually, we do this by calling an object's methods. The method responds, then control returns to the caller. Provided we know what type of object we are calling, and have an explicit reference to it, all well and good, this is obvious and works. However, in many situations, we don't have this information- we either can't be sure what type an object is, or we cant be sure which object we need to call, or where it is. This is problem, since without communication, we won't get much done.

Data models and views

A data model is a representation of something. It might be simple, such as a single value representing somebody's height, say, or a complete 3D simulation of the world. Whatever the data model is, we usually wish to visualise it somehow. We do this by drawing a representation of the data model in a window. We might show a person's height as the length of a bar in a graph, for example. For many purposes, we can simply bury our data model within the window object that visualises it, but sometimes, this is a bit restrictive. In a 3D modelling program, for example, we may want to open several views of our model, showing it from three sides, with perhaps a fourth showing the rendered model in a perspective view. Let's imagine that our data model is implemented by some kind of object- not a window or anything concrete like that, but just some sort of abstract object. Our application allows us to open as many views of this data model as we wish. Through any one of these views, we can change the model- for example, by adding or modifying parts of a 3D structure. Naturally, we expect a change made to the model to be reflected in all of the views that we have of it- it would be a rather poor application that forced us to go to each window in turn and hit a "refresh" button. We want the update to be done automatically.

We could do this by giving the data model a list of windows. When the model changes, it goes through the list and calls the window's refresh methods. The windows update, all is well. However, how many windows can we have? Well, we may decide to allocate four "slots" to a data model for its views. Then, once we've opened four windows, we are stuck. OK, so make it a dynamic list. That gets around the limit on the number of windows alright, but suppose now we have a completely new type of object that wants to know when the model has changed- we're stuck, since the data model only knows about windows and has only one response to a change- to refresh them. Any anyway, why burden the data model with having to know how to refresh windows- the model's job is to model! We are burdening this object with functionality that really has nothing to do with it- which goes against the grain of using OOP in the first place.We need something more general, and more OOP-like.

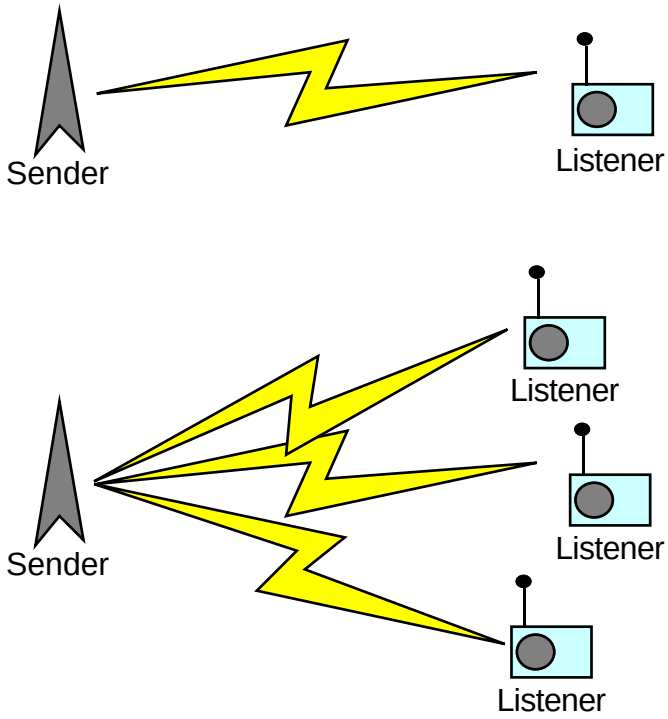
Comrades

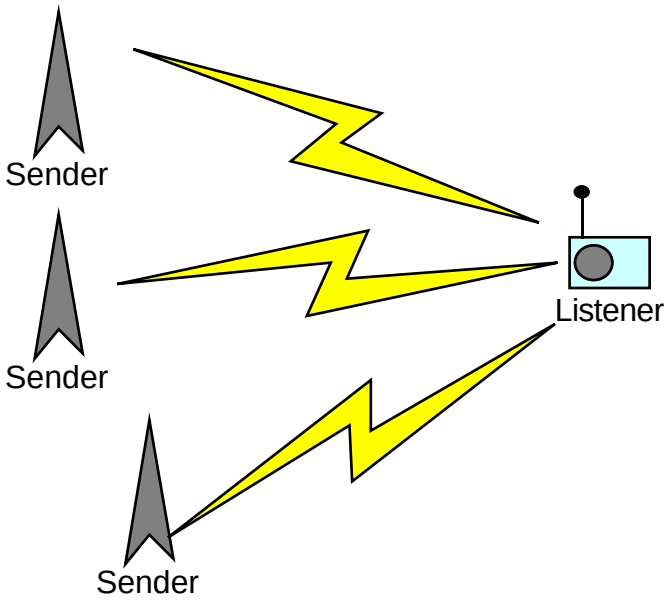
ZComrade is the answer. This is a very small MacZoop class that knows nothing much, except how to talk to other ZComrades. Most other MacZoop classes are ultimately derived from ZComrade, so most MacZoop objects are able to talk to any other. The only thing they need to know about the other object is that it is a ZComrade- whether it's a window or something else entirely really is not important, so we've got around the type problem. ZComrade can handle any number of comrades, so it gets around the numbers problem, and once the links are established, we no longer need the original reference to the object, so the whereabouts of the object is also of no interest. ZComrade also self manages its connections, so as objects come and go, no dangerous dangling pointers and stale references are left around to cause potential crashes.

The name ZComrade is a trifle misleading. The name was originally chosen to try and reflect that the objects cooperated, but in hindsight, a better name would have been ZTransceiver, since what these objects do is broadcast on one “channel”, and listen for broadcasts on another. The object that sends the message to the others is called the ‘Sender”, and the objects that respond to messages are called the “Listeners”.

In our example, we can make the data model a sender, broadcasting a “I have changed” message as needed. We make each view a listener of the data model. When the window receives the ‘I have changed” message, it responds by performing a refresh. Now, notethat the window decides what to do, not the data model. So when we add a new listener which is not a window, it can respond in a completely different way to the message “I have changed”. That’s object oriented!

many different topologies are possible:





• A single sender may have a single listener
• A single sender may have many listeners
• A listener may respond to many different senders

Setting up the channels

Channels are one-way. The listener responds to the sender. The sender broadcasts to its listeners and does not know or care what they do. If you have a pair of objects that have to communicate in both directions, you have to create two channels, one in each direction.

To connect an object to another, you call its ListenTo() method. This method takes a single parameter, which is the object to listen to. We need the object reference to set up the link, but not subsequently. Example: we have two objects, Smith and Jones. Here's how we could connect them:

1. Jones wants to receive messages from Smith	ListenTo(smith);
2. Jones wants Smith to listen to him	smith->ListenTo(this);

Here, Jones does the setup, but equally, Smith could do it. Or, a third party could do it, if it knows about the two objects. This has the advantage that neither object has to know about the other at any time:

```
jones->ListenTo( smith );
smith->ListenTo( jones );
```

Once the channel is set up like this, any messages that the sender transmits will be heard by all of its listeners. Remember, you can add any number of listeners to a sender.

Suppose we have a channel set up between Smith and Jones, but Jones is deleted. What happens if Smith sends another message? Well, don't worry. ZComrade makes sure that any channels are

automatically cleaned up when objects are deleted.

Closing the channel

We don't have to wait until an object is deleted to break the communication channel. At any time, we can call the StopListeningTo() method. This undoes the action of the ListenTo() method. Easy!

Sending messages

To send a message, the sender's SendMessage() method is called. Let's look at the prototype for this method.

```
virtual void    SendMessage( long aMessage, void* msgData );
```

This takes two parameters, the message itself, which is a 32-bit number, and an untyped pointer. As with commands, th emessage is just a number, though it is good practice to define some contsant for your messages so that your code is easy to read. It is also common to use four-letter codes for messages, to help ensure they are unique. MacZoop uses quite a few messages of its own internally, which is useful if you want to listen in on the activities of certain objects. So that we don't accidently introduce bugs, as with commands, MacZoop reserves all four-letter codes consisting only of lower-case letters (or all lower case plus a number) for its own use. Thus your own messages should have at least one upper case letter, or be simply numeric in nature, to avoid any potential conflict.

The <msgData> parameter allows you to pass useful information along with your message. Because this is untyped, it can be anything you want! It's usual to pass the pertinent information in the msgData- for example, not only what changed in the dat amodel, but its new value. This is done for efficiency- it saves the listener having to come back and ask for the data.

Responding to messages

To respond to a message, you override the ReceiveMessage() method, then look for the messages you are interested in. This is analogous to handling commands. Here's the prototype for the ReceiveMessage() method:

```
virtual void    ReceiveMessage( ZComrade* aSender, long theMessage, void* msgData );
```

You are passed three pieces of information: The sender of the message, the message itself, and any data passed along by the sender. Usually, your ReceiveMessage method will take the form of a switch statement, if you can respond to more than one message. Here's a typical example. We have a window that can respond to two messages. It responds to the "I have changed" message by refreshing, and to the "flush to disk" message by doing a Save. These are just hypothetical examples- they do not correspond to real messages in MacZoop.

```
void MyWindow::ReceiveMessage( ZComrade* aSender, long theMessage, void* msgData )
{
     switch ( theMessage )
     {
          case msgIHaveChanged:
               PostRefresh();
               break;

          case msgFlushToDisk:
               Save();
               break;
     }
}
```

Note that unlike HandleCommand, there is not usually any need to pass unhandled messages to the inherited method, since there is no chain apart from the one you set up using ListenTo, and the original methods do nothing at all.

Some pitfalls

When a message is sent, each listener in the chain is called in turn. The sender takes no special steps between each turn to save or restore graphics contexts, or even make sure that the message data has not been modified. Therefore listeners need to be careful! If your listener responds by doing some graphics, it should take care to save and restore the original graphics context. Remember- the sender might have been doing graphics elsewhere at the time! If you don't do this, the situation could arise where drawing in one window suddenly carries on in another one, because the first drawing sent a message, and the listener- the second window- changed the graphics context! Doh! Since the listener only knows how it is responding to a message, it is the listeners responsibility to take care to preserve any contexts, etc. that it establishes. The second thing is also fairly obvious- the message data sent should be considered read-only, since any listeners further along the line will get bad data if you damage it.

Incessant chatter in MacZoop!

Many MacZoop objects already send messages about their activities. For some application designs, you can take advantage of this and become a listener of these built-in messages. A typical example is ZArray. ZArray is used to store lists of related data, but the internal activities whn adding, removing or changing the positions of items in an array are all reported via messages. It would therefore be very easy to create a window class that displayed the contents of the array as an on-screen list, and the order would stay correct when the array was sorted, or extended, for example.

Another MacZoop object that chatters away like this is ZWindowManager. This gives regular reports about windows coming and going as the user interacts with the application. This is taken advantage of by Inspectors so that they can track the application's context dynamically.

Comrades and the command chain

It is very common that an object's boss wants to receive messages for its underlings. In fact, this is so common, it happens automatically. ZCommander overrides SendMessage() so that not only is the original list of listeners sent the message, but the boss is too, even though no explicit ListenTo() was ever called. In fact the boss is given the first opportunity to respond to a message, so is priveleged indeed!

This is useful in a number of situations. For example, a window opens a dialog box to set up some data structure. When the user clicks OK, the data needs to be extracted from the dialog box and used where it is needed, usually in the boss. So the boss window responds to the dialog closing message. This is covered in a lot more detail in the chapter about dialogs.

Note that while an object's boss always receives any messages that it sends, this process does not continue up the command chain. This reflects the purpose of the message system- it is to keep relevant objects informed. Usually, the boss of your boss is not interested in what you have to say, so these messages do not propagate up the command chain automatically. In fact, if you find yourself needing to tell your boss's boss something, it may indicate something wrong with your application design. Remember, if you need to link up to your boss's boss, you only need to call ListenTo and make it a listener.

Containers- Memory, Arrays and Lists

Much programming is concerned with collections of things. You may already be familiar with array types in C, that the stuff with the square brackets. Arrays are important and useful, and MacZoop provides some classes to implement them efficiently. However, before we talk about arrays and lists, let's look at plain vanilla memory.

Memory- computers these days have lots of it. From the programmers point of view, memory is just a block of bytes used to store stuff. We usually help ourselves by typing the memory so we know what's in it. Memory comes in many forms- on disks such as hard disks, CDs and floppies, and inside our computers, as RAM. We are not so much concerned with the physical aspects of memory however, so much as how it is divvied up logically. Programming is sually concerned mainly with the memory in the machine, the RAM. We also use disk memory, but these are usually treated in terms of files and this doesn't concern us here. Logivally, RAM comes in two forms, the STACK and the HEAP.

Stack & Heap

The stack is the place our temporary variables live. The variables we declare with in a function all go on the stack. This grows and shrinks as needed as a program runs. It doesn't require any explicit management. The trouble with the stack is that it's not a gret place to put data we want to keep around for a while. That's where the heap comes in. Memory that is allocated using new, or NewPtr, or NewHandle, comes from the heap. This is just a large pool that is ours for as long as the application runs. When we quit, the heap is returned to the system. Memory management of the heap is not automatic- we have to do it. That's the price we pay to have stuff that hangs around as long as we want- only we know when we're done, so we agree to take responsibility for its disposal.

Traditional arrays in C- the square brackets stuff, usually go on the stack. We have to decide how much storage we want when we program the application, and that's what we get, no more, no less. This is fine for some things, but not too great for others. We don't know how many windows the user might open in advance, so we can't simply declare an array of windows once and for all. What would be great is an array whose size can be decided on the fly, and stick around as long as we need it. Such a heap-based dynamic array would be great. Unfortunately, the C or C++ language does not provide such a beast. Luckily, MacZoop does, and it's called ZArray.

ZArray

You can store pretty much anything in a ZArray, but you do need to know the size of the items to be stored, and each item must be the same size. At first, this looks like a restriction, but it isn't really. The great thing is that we don't need to know how many items we'll want in advance. ZArray is a MacZoop object, based on ZComrade like many others. Internally, it stores its data in a Macintosh Handle, so it is using heap memory. When we create it, we tell it the size of the items we want to store. This can often be done using the sizeof() operator, which gives the size in bytes of a particular data type.

We can add new items to the array by calling its AppendItem() method. The array grows as needed, up to the available memory. We can retrieve ietms using the GetArrayItem() method, and put them back with SetArrayItem(). Once we're done, we can delete them using the DeleteItem() method.

As with classic C arrays, each item in the array has an index- it's position in the array. C Arrays start at 0 but MacZoop's ZArray starts at 1- there is no item 0. To find out how many items there are in the array, call the CountItems() method. This gives us one common way to iterate through all of the items in an array- using a for loop.

```
for( short i = 1; i <= array->CountItems(); i++ )
{
     array->GetArrayItem( &temp, i );

     // do stuff with the item....
}
```

Sorting

Because ZArray is an object, not just a reserved piece of memory like a classic C array, we can build intelligence in to the object that would have to be external in the classic world. A good example of this is sorting. ZArray knows how to sort itself- you just need to supply the basic function that compares two items and decides which comes first. This function is known as a comparison function, and takes a simple form:

```
typedef short (*SortCmpProcPtr)( void* itema, void* itemb, const long ref );
```

The two items are passed in, and we have to decide which comes first, by whatever means (we know what the data is, so comparison should be easy enough). If a comes before b, we return -1. if a comes after b, we return 1, and if they are the same, we return 0. The ref parameter is usually not important- it's there as a way to pass extra information to the comparison function if you need it.

ZObjectArray

It's very useful to use ZArray to store other objects. In fact, so useful and so common that we have a special class for it- ZObjectArray. This is a subclass of ZArray, but because we know it is storing objects, it can be optimised and made more efficient. Furthermore, it allows us to set up ZObjectArray as a template class, so we take advantage of better type checking, and avoiding typecasts. MacZoop makes considerable use of this class for its own needs, storing lists of windows in the window manager, lists of listeners in ZComrade, and so forth.

The best way to learn about these classes is to study the Class Reference.

Loose Ends- Timers and other knick-knacks

This chapter covers all the odds and ends that make MacZoop useful, but don't fit in elsewhere, or are too small to deserve their own chapter.

Timers

Timers are incredibly useful. You can attach a timer to any ZCommander object and the commander will be fed a regular series of messages at the interval you choose. You can attach more than one timer, in fact as many as you want. MacZoop timers are objects of type ZTimer, but for efficiency and ease of use, they are used slightly differently to other objects.

Timers are managed for you, so usually you don't directly make ZTimer objects. Instead you call the global SetTimer() function, which sets up the timer for you. Timers in MacZoop currently are based on ticks as their unit of timing. A tick is about 1/60th of a second, so this is currently the smallest uniy of time you can request of a timer. Timers can be periodic or one-shot. A periodic timer fires regularly at the rate you set, whereas a one-shot timer waits for the required interval, fires once, then deletes itself. Timers operate in cooperation with the main event loop- if the event loop is temporarily stalled for some reason, timers will not be able to fire and so accuracy will suffer. Bear this in mind- timers do their best to maintain regular time, but do not guarantee it.

Every timer has a unique ID number. This is provided so that if you attach multiple timers to an object, you can tell which one is calling you when it fires.

Using Timers

To create and attach a timer, use the SetTimer function:

```
ZTimer*         SetTimer(  ZCommander* aCmdr = NULL,
                           long id = 0,
                           unsigned long interval = kOneSecond,
                           Boolean isOneShot = FALSE );
```

Note that this function has four parameters, but all have default values, so you can call SetTimer with only the parameters you need- perhaps none.

<**aCmdr**> is the commander object that the timer will call when it fires. Note you can only attach a timer to a commander object, but in practice this means any window, the application, dialog item and many other types of object.

<**id**> is the timer ID number you wish to use to identify the timer. If you pass 0, a unique ID will be assigned. To find out what the ID number assigned is, call the returned object's GetID() method. ID numbers you assign only need to be unique to the particular commander they are attached to.

<**interval**> is the time period you require, in ticks. Therefore one second is 60, etc.

<**isOneShot**> sets whether the timer is periodic or one-shot. The default is false- i.e. a periodic timer.

The function sets up the timer and returns the object it created, if you need it. Your commander will start to receive messages from the timer as soon as the first time period elapses after the SetTimer call (as long as the event loop is entered). There is no need to prime the timer.

Handling timer messages

The timer message is received by the DoTimer() method. You can override this to handle the timer message. Here's its prototype:

```
virtual void    DoTimer( long timerID );
```

All you are passed is the timer's ID. This allows you to distinguish between more than one timer if you attached more than one. If you need more information, you will need to keep the reference to the original object returned by SetTimer.

You can stop your timer from calling you by calling KillTimer(), passing the ID number and the commander it is associated with. The object will be removed from the timer queue and deleted. You are not required to kill your timers before you are deleted- that will happen automatically.

MacZoop endeavours to give timers enough time to fire as regularly as possible. Since timers are cooperative, if you hog the CPU, your timers will temporarily stop. If you use a standard progress dialog for your lengthy operations, timers will be given time, so timers are an easy way to set up multiple cooperative tasks in an application without needing threads. If you want to keep timers going and you are unable to use a progress dialog, call the TimerTimer() function at regular intervals. Timers recognise certain types of commanders such as windows, for your convenience. If your commander is a window, it will Focus your window before calling your DoTimer method. However, it does not restore the previous focus, so beware! (Since timers fire in the main event loop, this should be a time where no graphics operations are otherwise occurring. You can use your timer message to do graphics of course.)

Uses for timers

Timers are very useful. They can be used to provide a simple way to have multiple parallel processes going on, without recourse to threads or other tricky techniques. Each time your timer fires, do a little bit more of a given process. You can also use timers to drive state-machine type logic, very useful for games and many other applications. You can do graphics in response to a timer- draw a bit more of a complex object, or show the next frame in animation, for example. Finally, you can use timers to time things!

Error Handling

Eventually, all applications, no matter how well written, will run into problems. Computing resources are finite, so at some point, you will run out of memory, hard disk space, etc. You need to accept this fact right fromthe beginning, and plan for it! Luckily, MacZoop makes error handling easy- or rather, C++ does, and MacZoop takes advantage of it.

Many toolbox routines return error codes. In other places, a NULL value of a pointer or handle is used to communicate an error. However an error is caused, you need to be able to detect it. MacZoop provides a number of functions that help you detect errors easily. These are implemented in the file ZErrors.cpp, and despite appearances, this is not any sort of object. These functions are just functions. Here's a few examples:

```
void FailNIL( void* aPtr );
void FailOSErr( OSErr theErr );
void FailMemError();
void FailResError();
```

These function are designed to be used on their own, or to "wrap up" a function that returns an error code, or a pointer value. All generate exceptions.

Exceptions

MacZoop error handling is based on exceptions. An exception is just an abnormal condition that aborts the normal flow of execution of some code, and transfers attention to an exception handler. This sounds complex, but isn't really. C++ provides a special syntax for this, the try/catch block. Here's one:

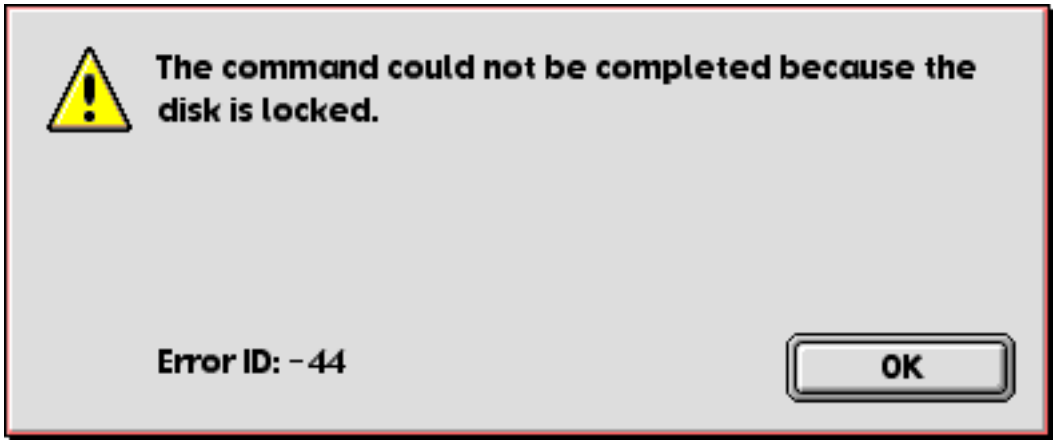
```
try
{
     DoSomething();
     DoSomethingElse();
     FailNIL( aHandle );
}
catch(...)
{
     CleanUp();
     throw;
}
CarryOn();
```

The code within the try {...} part of the block is the normal path of the code- it wold be executed even if the try was not there. If all is well, the code there is executed, then flow passed out of the try block, and continues after the catch {...} block, with the CarryOn() function in our example. If an exception arises in any part of the try block, including within any function called, flow is immediately passed to the catch block. Code here can attempt to rescue the situation, or else give up, perhaps cleaning up the internal state of the machine as needed. After executing the catch block, the program will normally continue with the line following, but in fact this is not usually what we want. Instead, we'd prefer to PROPAGATE the exception- that is pass it up to the next catch block, etc. Thid is an important point to note- try/catch block are NESTABLE. An exception transfers control to the nearest catch block (at the level of the exception), but this may be buried several levels down inside some subroutine. After cleaning up locally, we propagate the exception to the next catch block, then the next, so the complete execution state is unwound and cleaned up- this is a nice and easy way to make sure that we are dealing with our errors correctly,

and not leaving the machine in a bad state which could cause further problems later on. The C++ keyword throw is used to pass control to the next catch block, and our error functions all call it.

Error messages

When an exception reaches the the top level of catch blocks, which MacZoop arranges to be within the main event loop, an error alert message can be displayed to the user, explaining what happened, and hopefully, how to put it right. So that the right message can be displayed in a meaningful way for each error, errors have codes. This is a 16-bit number of type OSErr in MacZoop. MacZoop matches the error code to a resource whose type is 'Estr' and whose ID is the same as the error code. The Estr resource contains a short piece of text that describes the error and a possible fix. Here's a typical error message:



Normally, MacZoop begins all error messages "The command could not be completed because", then appends the Estr string. If no Estr is found to match, the generic "an error occurred." is appended. This Estr mechanism is common to a number of frameworks, so MacZoop can use Estr resources imported from MacApp, etc.

Silent Errors

Sometimes, you want to generate an exception that will not result in the user seeing anything. This makes sense sometimes where you can use the exception mechanism to make your code easier to write, but the situation is not a true error. A typical example is when the user presses a Cancel button on a progress dialog- this is expected to abort the operation, but is not really an error- the user knew what they were doing! These types of errors are known as 'silent' errors. MacZoop reserves a number of error codes as silent errors, including userCancelledErr and kSilentErr.
The error display mechanism is dealt with by the ZApplication method HandleError(), which is overridable- if you have a special requirement, you can display errors any way you want.

MacZoop exception blocks

All MacZoop errors are typed as OSErr. You intercept these using the following construct:

```
try
{
     ...
}
catch( OSErr err )
{
     ...
     throw err;
}
```

It’s also possible to catch other exceptions that you don’t know the type of. These may arise in library code where other exception types may be used. You can catch other types of exception using catch(...) Only OSErr type exceptions are handled by the HandleError method in ZApplication. Untyped exceptions are caught but can’t be further resolved, so the user will, if this occurs, see an error 998 displayed.

You should design your code to use try/catch sensibly, and be ready to deal with errors. This is especially important when dealing with disk files, or memory, both of which WILL run out sooner or later!

Cursor Handling

MacZoop provides a simple way to support cursors, colour cursors and animated cursors. For static cursors, you will usually set these when handling the AdjustCursor() method in your window objects. To set a particular static cursor, call the SetCursorShape() function. This is part of CursorUtilities.cpp. By using SetCursorShape() rather than the Mac OS toolbox directly, you gain a number of advantages. First, it’s just one call instead of two for the toolbox (GetCursor/SetCursor). Second, it will automatically use a colour cursor if one is available with the same ID, and finally, it works around some known issues with the toolbox which can cause crashes on some systems.

Animated cursors

Animated cursors are useful to show that the system is still working while a lengthy operation is performed. MacZoop uses the common VBL interrupt technique to animate cursors, and comes with three built-in- the watch, the beach ball, and the arrow with beachball. The advantage of VBL-based cursors is that once you start them animating, they continue without requiring further time given over to driving them. The drawback of this approach is that sometimes they will continue to animate even if your application crashes!

To start a cursor animating, you call the SetWatchCursor(), SetBeachBallCursor() or SetBusyArrowCursor() functions as required. Once started, the cursor will contnue to animate until the main event loop regains control, at which point, the animated cursor will be automatically cancelled. Therefore you only need to set the cursor at the start of your lengthy process, and it will stop as soon as you’re done and return to the main loop.

You can temporarily pause an animating cursor and set a static shape with PauseCursorAnimation(), and restart it using ResumeCursorAnimation(). This is done automatically when the application is suspended.

Adding your own animated cursors

If you want to add an animated cursor yourself, create the ‘CURS’ resources for the cursor, and an ‘acur’ resource which describes the frames of the animation. At runrime, you set up the cursor when the application initialises by calling InitAnimatedCursor(), passing the ID of the ‘acur’ resource. You get back a handle to the cursor data, which you can assign to a global variable. To start this cursor animating, call the StartCursorAnimating() function, passing your animated cursor handle and a time period. You should take care to check that no other cursor is animating by looking at the state of the gCursorTask variable. The best way is to write your own StartxxxCursor() function, modeled on SetWatchCursor(), but using your own handle.

Other Utilities

MacZoop comes with a handy grab-bag of functions which are there to help you, and are often used by other parts of MacZoop itself. These are not objects, and are not really related in any way. They are implemented in ZoopUtilities.cpp.

String Manipulation

MacZoop, like the Mac toolbox, uses pascal style strings quite a lot. To help use these, there are a couple of string manipulation functions.

```
void      CopyPString(    ConstStr255Param srcString,
                          Str255 destString );
void      ConcatPStrings( Str255 root, ConstStr255Param append );
void      CopyPStringTrunc(    ConstStr255Param srcString,
                               Str255 destString,
                               unsigned char ccLim );
void      ConcatPStringsTrunc( Str255 root,
                               ConstStr255Param append,
                               unsigned char ccLim );
```

CopyPString copies one pascal string to another.
ConcatPStrings joins a pascal string onto the end of another
CopyPStringTrunc copies a string, but only up to a limited number of characters
ConcatPStringsTrunc joins one string onto another, but limits the number of characters

```
void      CopyCToPString( char* cStringIn, Str255 pStringOut );
```

Converts a C-style string to a pascal string.

```
void      RealToString(   const double num,
                          Str255& str,
                          short decPlaces = 3 );
```

Converts a floating-point numeric value to its pascal string representation, accurate to the number of decimal places requested.

```
void      MacZoopVersionStr( Str255 aStr );
```

This returns a pascal string indicating the current version of MacZoop. You may want to use this in about boxes, etc.

Graphics Utilities

```
Boolean    IsColourPort( GrafPtr aPort );
void       SetPortBlackWhite();
void       SetHiliteMode();
short      GetMainScreenDepth();
```

IsColourPort() tests whether a given grafPort is colour or black and white. It return TRUE for colour.
SetPortBlackWhite() sets the current colours of the current port to black foreground and white background. This is a necessary step quite often before using CopyBits() or other bit-copying functions.
SetHiliteMode() sets a flag so that the next Invert or Xor drawing operation will use the highlight colour instead of literally inverting the pixels.
GetMainScreenDepth() returns the depth of the main monitor in bits, i.e. 1, 2, 4, 8, 16 or 32.

```
void       CentreRects( Rect* refRect, Rect* theRect );
void       Scale2Rects( Rect *theRect, Rect *refRect );
```

CentreRects offsets <theRect> so that it is centred over <refRect>. Neither rect's size is changed.
Scale2Rects changes the size and position of <theRect> so that it is completely contained within <refRect>. However, the original aspect ratio of the rectangle remains as it was. One handy use for this is creating thumbnail images from a larger original that have to fit in a particular space.

```
void       FrameGrayRect( Rect* aRect );
void       EtchGrayRect( Rect* aRect );
```

These functions can be used to draw 3D effect rectangles. These days, the Appearance routines accomplish this, but these can still be useful if your application is intended to run on earlier operating system versions.
FrameGrayRect draws a 1-pixel outline just OUTSIDE <theRect> which gives the appearance that the rect is sunk slightly into the surface. It calculates the colours required from the current port's fore and back colours.
EtchGrayRect draws a 2-pixel thick box in two colours to give the appearance that the surface of the window has been etched into. It is commonly used to draw group boxes, etc in dialogs.

```
void       ShiftPattern( Pattern* aPat );
void       ShiftCPattern( PixPatHandle aPat );
void       AntsRegion( RgnHandle aRgn );
```

These three functions give you a simple way to create the famous "marching ants" effect.
ShiftPattern() alters a black and white pattern by shifting all of the rows one place, and wrapping the top row to the bottom. ShiftCPattern() performs the same trick on a colour pattern.
AntsRegion() is intended to be called repeatedly to create the animation. It sets up a striped black and white pattern, and uses ShiftPattern to animate it. The region passed is framed in the shifted pattern. This method includes its own timer so that youi can call it as often as possible to create a steady "marching ants" animation.

Memory utilities

```
Boolean    EqualMem( void* a, void* b, const unsigned long length );
Boolean    EqualHandle( Handle a, Handle b );
long       ChecksumHandle( Handle h );
```

EqualMem can compare two arbitrary tracts of memory to see if they contain the same data. It returns TRUE if they match, FALSE otherwise. It ios inadvisable to use this function to compare very large pieces of memory, because if they are the same or very similar, it can take a long time to perform the test. EqualHandle() provides the same facility on any two handles. CheckSumHandle() can be used to generate a checksum for the data in a handle. This can be handy to identify or recognise the particular contents of a handle, without having to examine the data completely. The checksum is the modulo-11 weighted sum of the data bytes of the handle.

```
Handle     GetDetachedResource( ResType aType, short anID );
```

Loads, detaches and returns the handle to the resource requested. This is equivalent to GetResource() followed by DetachResource().

```
short      GetRandom( short min, short max );
```

Retuns a random integer between min and max inclusive.

```
long       SquareRoot( long n );
```

Returns the approximate square root of the argument, but does not use MathLib. This is handy where you need a rough and ready square root calculation, to the nearest integer. Used by Mac-Zoop for the window tiling command.

Using the Notification Manager

```
short      NotifyAlert(    const short alertID,
                           NTAlertFlags flags = ntAlertDefault );
```

MacZoop does not come with any specific alert handling functions, so usually, if you want to show an alert, you can just use Alert(), etc as usual. This function is handy if you want to show an alert if the application is active, but post a notification manager alert if it's in the background. This function both hides the details of the Notification manager from you, and deals with the correct behaviour required when the user resumes your application to deal with the alert.

You call NotifyAlert(), passing the alert ID of the alert. If the applicaiton is in the foreground, the alert is shown. It acts just like the toolbox function Alert(), returning the item the user chose. If the application is suspended, the alert is not shown. Instead, a notification manager request is put together and posted, then this method sits in a loop, processing events. It will not exit this loop until the user resumes the application, at which point the original alert will be shown, and the user can respond as normal. The <flags> parameter is used to tell the function what features to add to the notification:

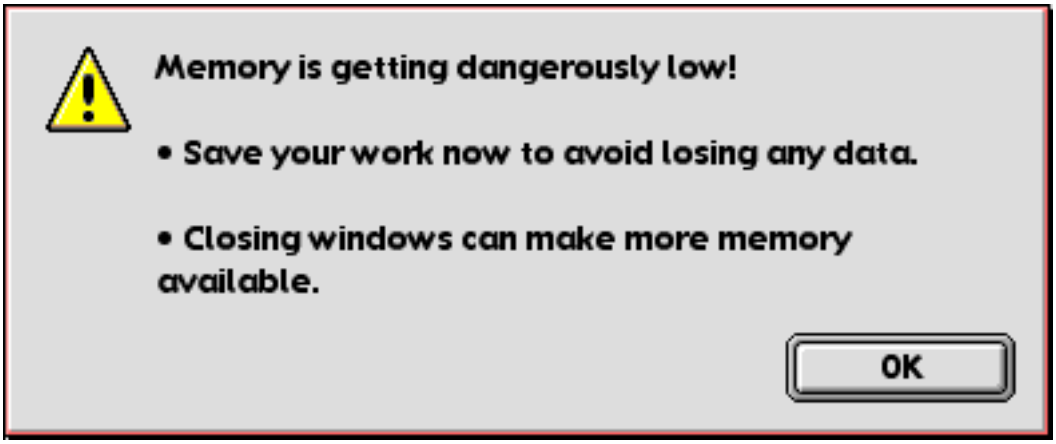
ntAlertPlaySound	notification manager beeps
ntAlertDisplayMessage	notification manager puts up alert with app name "requires atten tion"

These flags can be added together, or ntMinimalAlert passed to just get the flashing icon. NotifyAlert sets up the icon and attention message from other MacZoop information, so will always generally be correct. This is also used by the default HandleError() method.

Memory Management

MacZoop provides a convenient mechanism for helping to make your application more robust and tolerant of memory problems. When it starts up, every MacZoop application allocates a "shortage fund" which it can release in part or in whole when memory becomes tight. When it does this, it provides the user with a warning. This is intended to give them time to save their work before things get any worse. You can also hook into this mechanism to release memory you know is no longer required (though doing this requires that you somehow track your memory in such a way that it permits this kind of decision).

The size of the shortage fund is usually 64K. This is set by the value of <kShortageFundSize> in ProjectSettings.h. Usually, 64K is fine, but of course you can change this if you want. When the Mac memory manager cannot find the memory it requires, it calls back ZApplication which will release as much of this fund as needed to satisfy the request. At the next event time, MacZoop will try to replenish the fund. If it fails to do so, the user sees the warning message:



ZApplication will also grey out the New and Open commands until the problem has been resolved, since those commands are usually the ones to make such a situation a lot worse! The purpose of this mechanism is to give the user time to save their work and back away from the problem situation before things get so bad that a crash is likely. You should not rely on this mechanism to absolve you of proper memory management and exception handling- it's there purely as a safety net.

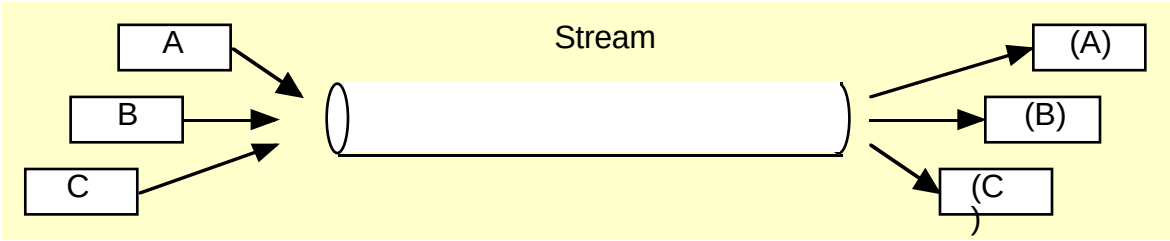
To extend the use of this mechanism, you can override ZApplication's MemoryShortage() method. Here you can release further memory use of your application. You should only release memory- you should not display windows or throw exceptions here, because this method is called from within a toolbox callback.

MacZoop Streams

MacZoop streams are a powerful way to implement temporary or permanent storage of data in a simple to use way. While powerful, streams also have their pitfalls. This section explains streams and how to use them, and also how to avoid getting into trouble with them.

What is a stream?

A stream can be thought of as a pipeline into which you can put data. Later, you can get the same data out, but only in the order in which it was put in. In this respect, a stream is a bit like a FIFO (first-in, first-out) queue. However, streams are more than this. The actual storage medium of a stream is not specified- it could be a disk file, some local memory, or a remote server or other application. By using the polymorphism of C++, all streams work the same way regardless of how they store data, and thus you can use a single stream technique to save your data in many different forms. It is possible to envisage implementing file saving, cut/paste, drag/drop, undo and networking with a single pair of methods in your application!



What can be stored in a stream?

Really, anything can. At some level, a stream is just an array of bytes. However, to make it easier to use, MacZoop provides an API to the stream that allows a higher-level approach to be taken- you can store single characters, up through short words, long words, Rects, Points, Handles, arbitrary data arrays and even entire objects in the stream. This last point really indicates the power of streams- you can save whole objects to a stream (maybe a filestream), then later on read them back in exactly as they were when they were saved. You can pass objects from one application to another over a network stream, or place them on the clipboard or in an undo task using a handle stream.

MacZoop streams

MacZoop implements two basic forms of stream- ZFileStream and ZHandleStream. As their names suggest, the former stores its data in a file, the latter in memory. They both derive from a new abstract class ZStream, which defines the stream API independently of the storage medium.

In order to allow whole objects to be passed through a stream, a common streamable object is required called ZObject (an abstract class) from which many other (but not all) MacZoop classes are descended. Only objects decended from ZObject can be streamed Most significantly, ZComrade inherits from ZObject, which means that the majority of MacZoop objects are streamable. Many standard classes such as ZComrade, ZCommander, ZWindow and ZScroller are streamable already. Some other ZWindow derivatives are also streamable too, such as ZPictWindow and ZGWorldWindow, and also ZGWorld itself- you can pass an entire GWorld

lock, stock and barrel through a stream. ZArray and ZObjectArray are also streamable- you can pass entire arrays or sets of objects to a stream.

Object References

Many objects refer to other objects. For example ZCommander has a reference to its boss, ZComrade has lists of its talkers and listeners, etc. In order to recover the correct relationship between objects when they are reconstructed from a stream, it is necessary to save a reference to these objects in the stream. This is where one of the main pitfalls of streams shows up. If you were to save the object's address in the stream, clearly this address is unlikely to be correct if it is read back from a file, perhaps days or weeks after it was first put there, or passed to another program on another machine. Instead, MacZoop saves enough information about the object to completely reconstruct it afresh every time it is read back in. Here, another problem crops up. In many situations, more than one object may refer to the same object- two ZCommanders may have the same boss, for example. Clearly, if the boss reference was saved to the stream twice, you would end up with two different objects when read in, not the desired situation. ZStream keeps track of this- for each object written to the stream, ZStream will check if it has already written the same object before (by comparing its address with a table of stored object addresses), and if so, write a special marker to the stream indicating that it's the same object as an earlier one. Only the first time an object is written to the stream is it saved in its entirety. When read back, the object is created from the stream, but then subsequent requests for that object will simply return the same address again.

In this way, ZComrade writes all of its talkers and listeners to the stream, and ZCommander its boss, without needing to worry about other objects writing the same references.

Some objects are never written to the stream, nor recreated from it- ZApplication being the notable one. It is important that there is only ever one application, so references to gApplication are written in such a way that reading it back never results in a new ZApplication object being built.

ZObjectArray saves all of the objects it contains to a stream on request, so you can save lists of objects as well as single ones.

Stream Hierarchies

Here is another pitfall of streams. Say you want to save a window to a stream. You call its WriteToStream() method which dutifully stores all of its internal state in the stream. However, a window is derived from a ZCommander so the first thing it does is call the inherited WriteToStream() method which writes the window's boss to the stream, which could be another window! So now you have two windows in the stream. ZCommander also calls ZComrade's WriteToStream method, which saves all of the objects that are listening to the window or who the window is listening to. Before long 90% of your application has been written to the stream! When you read the window back in, that 90% is reconstructed, which might be what you want, but it might not be. If the stream was a file saved by a visual application builder, that's great- a whole complex application is built with one line of code. However, if you just wanted to save the window to an undo task, that's probably not what you wanted.

Unfortunately, it's extremely difficult to deal with this problem generically, so MacZoop makes a compromise. The compromise is that an object's superclass is always written to the stream when

you use the WriteToStream() method, and read by the ReadFromStream() method, which is suitable for constructing the whole hierarchy from a stream, but if you want to stream only part of this structure or data content of a window, say, then you have to write your own streaming methods. Since both of the standard methods are never called unless you call them (they are not called by the framework), you have everything under your own control.

If you have followed this discussion, you may also have realised that there is another problem related to this. If you do not save the additional objects referred to when you stream an object, then you won't know what they are when you read the object back in. Thus it's possible to stream a ZCommander object for example, but lose all the talkers and listeners it may have inherited from ZComrade. Again, the solution to this is left to your own implementation- if the stream is a temporary one such as an undo task, it would be OK to save real object addresses in the stream, but this would not be a good idea for a file stream. Again, since MacZoop can't predict what you're trying to do, it does not make any potentially false assumptions and leaves the details to you.

Stream Order

As mentioned at the top of this document, a stream is a FIFO queue. This means that you get the data back in the same order as you put it to the stream. If you request a short where you originally wrote a long, you'll get a short, but the stream is now out of step and subsequent calls will retrive bad data. Thus it is vitally important that your write and read methods follow the exact same sequence and sizing as each other, and that this is maintained at all levels through a series of objects. If you want to write variable-length data, use WriteData or WriteHandle which store a length count in the stream to maintain synchronisation. If you want to write optional data, write a flag to indicate if the data is actually present or not, and read it back accordingly. It is possible to skip data when reading (and writing in fact, but this is rare), in general you should always read back what you wrote.

Note that ZStream takes care of writing NULL objects and Handles- you do not have to check for this before you write them, and you'll get NULL back when you read them.

Persistent Objects

The implementation of streams within the MacZoop framework is intended to provide persistent objects. That is, objects that can lie dormant in a file or other storage until they are "realised" in memory. This allows (amongst other things) one application to create objects on behalf of another- for example a visual interface builder can be used to define the user interfaces for an application to use at run time. To this end, the WriteToStream() and ReadFromStream() methods that are provided by ZObject and overridden by many other MacZoop classes are intended to provide persistent objects and as such substitute for the normal constructor and/or initialiser for the object. When such an object is encountered in the stream, MacZoop makes a completely new object of the right type and calls its ReadFromStream() method to initialise the object completely from the stream. There are no constructor parameters and no separate initialiser is called- the ReadFromStream() method recovers enough information to do the job completely. This is why MacZoop, by default, always calls its superclass's ReadFromStream() method, so the entire object right down to its root, is rebuilt.

In order that MacZoop can make objects of the right type, it is necessary to provide information to the stream about the object type, and provide some way for MacZoop to instantiate them. This

task is done by another new class, ZClassRegistry, of which there is only one stored in the global variable gClasses. Every streamable class is derived from ZObject, which provides a <classID> data member. This must be initialised by the object's constructor to a unique code for that class. All of the standard MacZoop classes derived from ZObject now do this. In addition to providing this ID, every streamable class must provide a 'Constructor Function' and be registered with gClasses. The constructor function is static and simply returns a new object of the desired class. ZApplication has been modified to register the standard streamable classes with gClasses. Your application can override ZApplication's RegisterClasses() method to register additional streamable classes- each class is registered by associating the class ID with the constructor function.

When an object is instantiated from the stream, it is looked up in the registry and the constructor function is called. Then the ReadFromStream() method of the new object is called to initialise it from the stream. All of this takes place automatically when the stream's ReadObject() method is called. Subsequent references to that particular instance in the stream return the same object, not new copies of it. For this to work the first object reference in the stream for a particular instance must be the actual complete object- again, this is all taken care of by the stream's WriteObject() method- in general you do not need to care how it works. Note that the 'constructor function' is expected to make a new object, but doesn't have to, as long as it returns a valid object reference. Thus references to <gApplication> return <gApplication> via the constructor function, and do not make new ZApplication objects. Your own code could use this technique if it makes sense for your application.

MacZoop reserves all class ID's consisting of only lower-case letters for its internal classes- your own class ID's should contain at least one upper-case letter. It is up to you to ensure that all class ID's are unique and that each class is registered only once.

Using Streams

To use a stream, you instantiate a stream object (currently ZHandleStream or ZFileStream), then save data to it by calling its Writexxx() methods, where xxx is some data type- char, short, long, Handle, etc. You cannot instantiate a plain ZStream object because it is a pure abstract class. The stream object may be created using 'new' or built on the stack. If a ZFileStream, you need to pass a filespec or filename in the constructor, just as you would to ZFile (ZFileStream multiply inherits from ZStream and ZFile). With a handle stream, you can pass an existing handle to the object, or allow it to make one for you. For an existing handle, you can specify whether you want the stream to take ownership of the handle (ownership means that the handle is disposed when the object is deleted).

Another way to employ streams is to use the WriteToStream() and ReadFromStream() methods derived from ZObject. Here, you make the desired stream object then pass it to one of these methods. Internally they write or read their data to the stream. The framework methods, as stated above, are intended for the implementation of persistent objects. For other uses, you might like to create your own stream encapsulation methods along similar lines.

Once all the data is streamed, you can discard the stream object. In the case of a ZFileStream, the file will be closed and will exist for later use on disk. For a Handle stream, you need to get the handle before discarding the stream (if the stream owns it) using its GetHandle() method. You can then pass this handle to any suitable routine, for example you can use ZClipBoard's PutData() method to store the data on the clipboard, or store the handle in an undo task, etc.

To read data from a stream, you instantiate the stream object, passing it a filespec if a ZFileStream, or the storage handle if a ZHandleStream, then call its Readxxx() methods to recover the data. The data must be read out in exactly the same order it was written in. You cannot determine the type of data stored at the next point in the stream- you need to know what to expect and normally this is done by arranging your write and read functions in matching pairs. Once read, the stream object can be discarded.

Important stream-related settings

For streaming to operate at all, the setting _MACZOOP_STREAMS must be set to 1. This #define is in the MacZoop.h file (NOT ProjectSettings). If this is 0, most of the streaming code is not compiled- this is done to save space for those applications that do not use streaming.

If you intend to stream objects of your own devising, you must set up the class IDs and Constructor Functions needed to create them from the stream. MacZoop provides a number of macros to make this easier.

To define the class ID for a given object, use the DEFINECLASSID() macro:

DEFINECLASSID(<classname>, <id>);

e.g. DEFINECLASSID(ZMyClass, ‘ZMCO’);

You must then assign the class ID to the data member <classID> in your constructor:

classID = ‘ZMCO’;

Note that class IDs are defined as OSTypes for convenience- it is usual that a four-letter code is used for these. All lower-case codes are reserved for MacZoop- your codes should have at least one uppercase letter.

To declare the constructor function for your class, use the CLASSCONSTRUCTOR() macro at the top of your .cpp file:

CLASSCONSTRUCTOR(<classname>);

e.g. CLASSCONSTRUCTOR(ZMyClass);

Remember that streamable objects must have a constructor defined that has no parameters.

All streamable objects must be registered at application startup. The ZApplication method RegisterClasses() is called, and you should override this to register your own streamable classes. Be sure to call the inherited method to register standard streamable MacZoop classes. To make registering your class easier, a single macro is used to generate the required call for each class:

REGISTERCLASS(<classname>);

This expands to set up the registry with your class ID, constructor function and class name.

[intentionally blank]

A C++ Application framework for Macintosh

created by Graham Cox

Class Reference

Edition 3, January 2000
For framework version 2.1

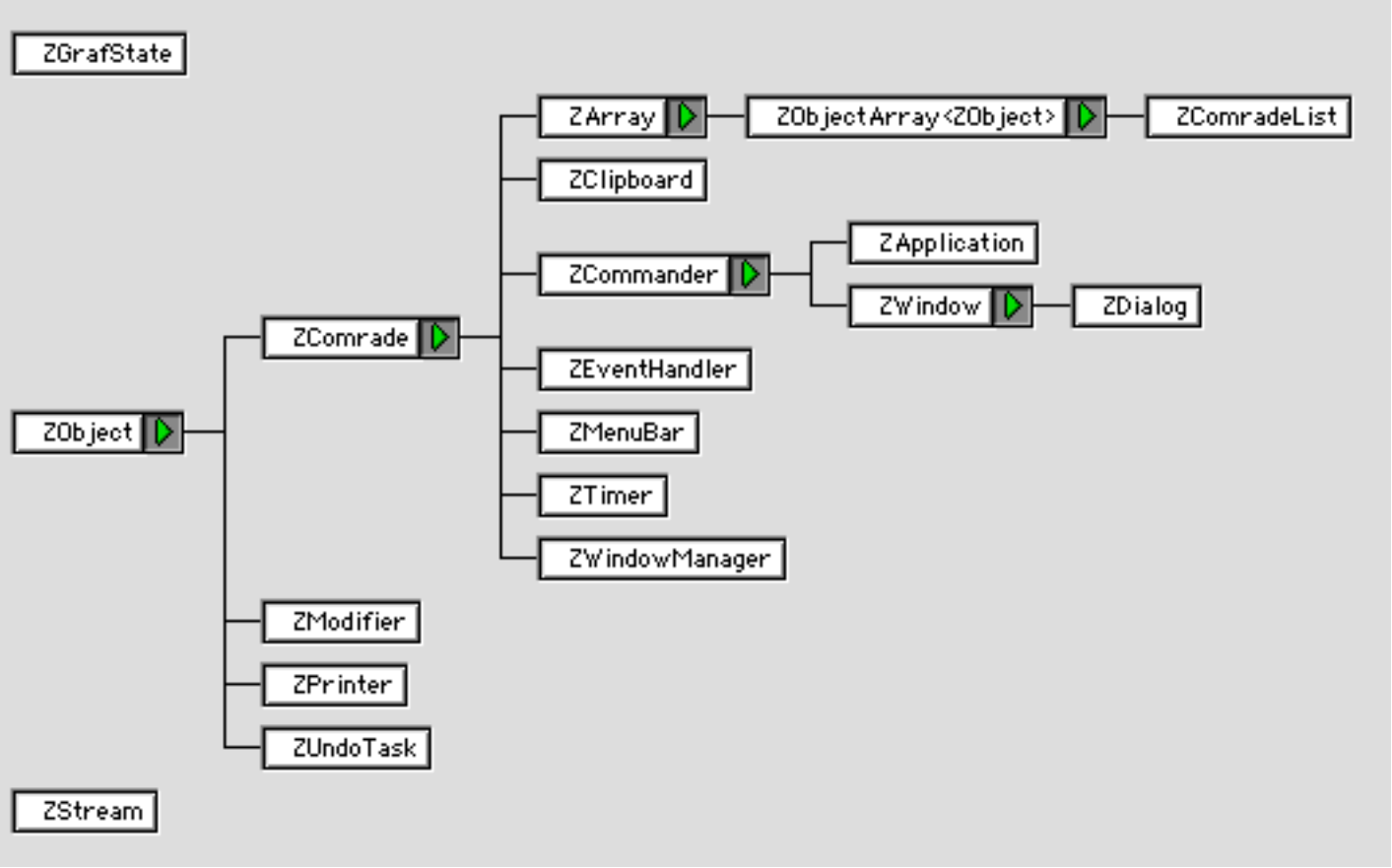
Introduction

This Class Reference is a complete breakdown of all of the classes and utility code in the Mac-Zoop 2.1 release. This gives a brief description of all of the methods and functions, but does not explain in detail how to use them. To get the most out of this reference, you should have read the first part of the manual that explains how the major pieces of MacZoop fit together, then you can explore this part to discover the finer points.

The classes and utility code is listed alphabetically by the name of the file that contains the code. The methods and functions are described in the order they appear within the class or file.

Classes Overview

If you have CodeWarrior 10 or later, choose “Show Hierarchy Window” from the Window menu to see how the different parts of MacZoop fit together- this is quite a useful way to learn your way around the framework. (You will need to have compiled the project successfully to see this diagram, and activate the browser).



This diagram is for the very basic MacZoop project that we worked with in the Getting Started chapter. For the Demo Project, the chart will be larger, including all of the additional classes that the project makes use of.
This chart shows only how the various classes relate to each other in terms of their definitions- what is a subclass of what, etc. It does not show how they are linked together in functional terms to make a working whole.

Global Variables, Macros and Switches

In MacZoop, global variables are not defined all in one place, but rather in the headers of the classes that they most suitably belong to. Thus it can be hard to see what is available at a glance. This lists all of the main framework globals with a short description of each.

Macros are a handy way to define inline code that is used frequently, as well as providing more readable labels for common values. Some of the more important macros are listed.

ZDefines.h

ForgetObject(). This is a macro that encloses the delete keyword, but also makes sure the variable passed is set to NULL, which can prevent errors by making sure that stale references do not stay around to cause trouble later. In general you should use this macro (which is designed to appear as a function) rather than calling delete directly.

ForgetThis(). This macro allows a heap object to delete itself. This should be done with care, since a self-deleting object precludes being used as a stack object.

GetZWindow(). Given a Macintosh WindowPtr, this returns the associated ZWindow object. This currently works by checking the value of the <windowKind> field, and if the value IS_ZWINDOW_KIND (772) is found, it casts the refCon to the object reference. In general you should avoid doing this if at all possible, since there are other ways to obtain the front window as an object, or iterate through the windows in the application. However, if you absoultely have to do this, using this macro will mean that you are safe from implementation changes that may be made in the future.

MIN and **MAX** are handy macros for returning the smaller or larger of a pair of values respectively. You should take great care passing functions that return results directly to these macros, since the expansion may not give you the expected result.

ABS is a macro that returns the absolute value of a number (i.e. the - sign, if any, is dropped). This can be cause problems with certain functions, so be careful.

CMP returns the relative magnitude of two quantities, -1, 0 or 1 depending on whether the parameters passed are equal or different. Can be used for sorting arrays based on numeric quantities, though this is not a comparison function in its own right.

SGN returns the sign of the argument- either 1 (positive) or -1 (negative).

Many non-printing keys on the keyboard have more descriptive labels that you should use to help make your code more readable. They are:

- TAB_KEY
- RETURN_KEY
- ENTER_KEY
- ESCAPE_KEY
- UP_ARROW_KEY
- DOWN_ARROW_KEY
- LEFT_ARROW_KEY

- RIGHT_ARROW_KEY
- BACKSPACE_KEY

There is a macro for the standard arrow cursor to add to the usual toolbox constants iBeamCursor, crossCursor, etc. It is ARROW_CURSOR

The current framework version can be obtained using the macro **MACZOOP_VERSION**. This returns a short formatted like the 'vers' resource number, using 4 bit fields. For example, version 2.1 returns hexadecimal 0x0210, or 528 decimal.

Globals

gApplication is the object reference of the application. Unlike some globals, this is not automatically declared as extern ZApplication* in the header for ZApplication. This is because this class is commonly subclassed and your code is likely to want to access it as the subclass type, for example <MyApplication> Therefore the inclusion of the required extern MyApplication* gApplication in your code is left up to you.

gMenuBar is the ZMenuBar object handling the main menu bar.

gWindowManager is the ZWindowManager object responsible for organising your windows.

gAppSignature is an OSType that defines the creator code for all files your application will create. This is initialised to <kApplicationSignature> which you can define in your 'ProjectSettings.h' file.

gIsAColourMac is a boolean indicating whether Colour QuickDraw is available. If you intend your application to run on black and white only Macs, your code may need to check this frequently to avoid making toolbox calls that are not implemented on those machines. Note that since MacZoop requires System 7.0 or later to run, many colour QuickDraw routines are available even on "classic" Mac hardware.

gMacInfo is a global tMacInfo structure that indicated the presence of a number of common Macintosh toolbox additions, and also the system version. This is sometimes more convenient than making calls to Gestalt().

The tMacInfo structure currently consists of:

```
typedef struct
{
     Boolean         supportsColour;          // colour quickdraw available
     Boolean         hasDragManager;          // drag manager available
     Boolean         hasFPU;                  // has a floating-point coprocessor
     Boolean         hasAppleEvents;          // has apple events
     Boolean         hasAppearanceMgr;        // has the appearance manager
     Boolean         hasQuickTime;            // has QuickTime available
     Boolean         hasImgCompressionMgr;    // has Image Compression Manager
     Boolean         hasNavigationServices;   // Nav Services installed & available
     Boolean         hasContextualMenus;      // Contextual menu services available
     Boolean         hasModernWindowMgr;      // MacOS 8.5 Window Manager or later
     short           systemVersion;           // current system version number
}
tMacInfo;
```

gCurHandler is the commander that starts the current command chain. Developers are advised that this variable may go away in a future version, since its abuse by third party code can (and has) caused instability in the past and is a common cause of less serious bugs. You should treat this variable (if you have to use it at all) as read-only and the property of ZEventHandler. Be aware that calling through it at certain times may cause problems if it hasn't been updated for the current context. Using undocumented techniques or ignoring this advice may result in an unnecessarily poor quality product, so please take care! It is in order to resolve these problems in a rational way in the future that this variable may be removed.

gClipboard is the global ZClipboard object.

gFontMenuID is the ID of a "Font" menu used in the main menu bar. If you call AppendStdItems() for a "Font" menu, this is set with the ID. Other classes that use a font menu, such as ZTextWindow, can use this variable to handle font selections easily.

CursorUtilities uses several global variables. You should not use them in your code- they are private to the cursor utility code. Use the documented API instead.

There are numerous globals scattered throughout MacZoop used to set up or handle toolbox callback procedures. In general you should not use these in your code- they are there so that you can use an object-oriented method to access callbacks. These include control action procedures, user items in dialogs, apple event handlers, drag manager procedures, list definition procedures, etc.

Project Settings

With every project you make with MacZoop, you have the opportunity to customise its behaviour in certain ways. The file "ProjectSettings.h" is where you can do this. The file supplied contains the default settings, but if you copy the file and make sure it replaces the standard one in your project, you can change them. Here are the settings and what they mean.

<**kApplicationSignature**> is a four-letter code (OSType) that is used to define the creator code for your application. It is assigned to the global variable **gAppSignature** and is used wherever a file is created. In addition, you'll usually set the creator code of your application itself to the same code. To do this you need to open the preferences for your project and set the creator type in the panel called "PPC Target" or "68K Target" as appropriate. If this matches <kApplicationSignature> and you have set up a 'BNDL' resource, this will ensure the Finder displays the correct icons for any files your application creates.

USE_SIGNATURE_FROM_BNDL sets whether your application will set the value of <gAppSignature> from the BNDL resource rather than hard-coding it. It should generally be ON.

CHECK_FREF_RESOURCE_TYPES allows the application's internal listof openable file types to be set from the 'open' or 'FREF' resources, rather than hard-coded. Usually, this should be ON.

<**kShortageFundSize**> sets the size of the emergency shortage fund that is used to help bail your application out of a memory crisis. It value, in bytes, defaults to 64K.

MAKE_UNTITLED_STARTUP_WINDOW sets whether ZApplication automatically makes a new “untitled” window when it is launched. Normally this is what you want for a typical application, so the default is ON. However, if you are using MacZoop to make a “droplet” type of utility, you might not want a window, and won’t have a “New” menu item. Setting this to OFF makes sure you don’t make a window. You can achieve the same effect by overriding MakeNewWindow() to do nothing, but this may be easier. Note that your application must be able to receive high level events as well for this to work- there is a flag to set this in the ‘SIZE’ resource.

_SLOW_BUT_SURE_DESTRUCTION affects the way that object clean-up is done when the application quits. Normally, this is OFF, since the entire application heap is freed in one go when it quites, which is faster than individually freeing every single object. However, some developer tools report this (erroneously) as an error. If you want to avoid such reports, set this to ON. You app is likely to take much longer to respond when quitting however.

PRINTING_ON sets whether your project supports printing, and therefor needs to have ZPrinter included. The default is ON. If you know you don’t want to print, setting this to OFF will allow your application to be slightly smaller.

_PRINT_USING_PROGRESS_BAR sets whether a progress bar is displayed when spooling pages. It has been reported that some printer drivers, e.g. Hewlett-Packard, will crash if this option is on, therefore the recommended setting is OFF (since you can’t predict what printer drivers your users have installed).

APPEARANCE_MGR_AWARE sets whether your application registers with the Appearance Manager and can use Appearance-style controls, etc. If OFF, then your application will not use any appearance functions, and will not require linking with AppearanceLib. Many parts of MacZoop will still use appearance-style controls using the toolbox remapping mechanism, and simulate the greyscale appearance for dialog items, etc. If ON, you need to link with AppearanceLib, but all controls, dialogs, etc will be appearance aware. Applications so built will still run correctly on systems without Appearance. This shoulkd generally be ON if you can afford it!

If you have turned on the above, you can also turn on **USE_PROPORTIONAL_SCROLLBARS** and **USE_PROXY_ICONS** if you wish. These allow MacZoop objects that use scrollbars to provide the proportional behaviour, and MacZoop windows to display a title bar proxy icon. These features only exist on Mac OS 8 and later.

_ALL_FLOATERS_ACTIVE is used to control the appearance of floating windows. The usual appearance is that all floaters are shown in the hilited (active) state. However, if you prefer the behaviour where only the topmost in the floating layer is active, you can set this to OFF (default is ON).

_ACTIVATE_EVENTS_ARE_REAL sets how activate events are sent to windows. Usually ZWindowManager and ZEventHandler ensure that windows get the activate events they need without going through the Macintosh toolbox. This helps keep things working well in a floating environment, hence the default is OFF. If for some reason you need to have real activate events that go through the Mac’s event queue, you can set this to ON. However, this is NOT recommended unless you absolutely have to.

_UPDATE_ON_SELECT controls the order of doing things when a window is dragged by its title bar. If ON (the default) the window is updated as soon as it is selected (made active) without waiting for an update event, which can give snappier performance. If OFF, the redraw is done when the update event finds its way back to the app, which is the more usual way Mac applications work. This event is not processed until the drag completes, so you get an non-updated part of a window visible on screen as long as you drag the window's outline, which is rather ugly- the behaviour was changed to solve this problem.

_DRAGWINDOW_COMPATIBLE also controls how windows behave when dragged. ZWindowManager reimplements a version of the toolbox call DragWindow() which works in a floating environment. In this reimplementation, things are done slightly differently which gives a faster, better responding behaviour than the Mac toolbox. In this case, this means that if the window is going to be selected, it is done at the start of the drag, not the end. To make ZWindowManager's behaviour match that of DragWindow(), set this to ON.

_ALPHABETICAL_WINDOWS_MENU if ON, (default is OFF) the Windows menu, if any, is kept in alphabetical order. By default it is kept in order of creation.

_ENUMERATE_WM_CMDS will automatically assign command-key equivalents command-0 to command-9 for the first ten windows in the Windows menu if ON. If OFF, no command-key equivalents are appended.

_USE_DIR_POPUP is used to switch on the useful feature where a command-click in the title bar of a window pops up a menu with the path to the file (if the window has an associated file of course). This functionality relies on third-party code called "Directory Pop Up" by Marco Piovanelli. A version of it is available in the More Classes->Goodies and Extras->Directory Pop Up folder.

_AUTO_MBAR_HIDING allows you to have a "pop-up" menubar. This is a rather unusual behaviour though quite useful for some applications such as presentation programs or games. If ON, the menubar will only be visible if the mouse is within its normal area, otherwise it is hidden. The default is OFF.

_CUSTOM_ICON_SUPPORT allows you to decide if you want custom icon support in ZFile. If OFF, you can save quite a bit of space and your project doesn't need "PixMapUtils.c" to be included. The default is ON.

_ZOOM_RECT_FX allows you to switch off the Finder-like "zoom open" and "zoom closed" rects when windows and dialogs are opened and closed. If ON (the default), your project needs to link to <DragLib> if a PowerPC project.

_WPOS_WINDOW_PLACEMENT allows the window save and restore code to be compiled in. Usually this should be ON. Switch it off if you are implementing your own window position saving code.

If the above is ON, you can also switch on **_AUTO_WPOS_FOR_DIALOGS** and/or **_AUTO_WPOS_FOR_FLOATERS**. This will allow all dialogs and floaters' positions to be saved in the prefs file automatically without you having to write any code to do this. You still need to create the prefs file however, since that is not automatically done.

_CANCEL_PROGRESS_THROWS_EXCEPTION. Usually, this should be ON. If you use a progress dialog, then when the user hits the Cancel button, an exception is thrown with the error code <userCancelledErr>. This is a silent exception. This saves you having to respond to the Cancel button yourself. However, if you prefer to do this, set this to OFF.

_USE_NAVIGATION_SERVICES. If ON, MacZoop will use Navigation Services whenever it needs to display the Open or Save File dialogs. This requires that your application links with NavigationLib. Applications so built will revert to Standard File on systems without Navigation available, so this should generally be ON.

_USE_NAV_SAVEREVERT_ALERTS, if the above is ON, you may also set this. This uses Navigation's built-in alerts for the "Save Changes/" and Revert alerts. However, this is actually not recommended, since these alerts position over the topmost window, which may be a floater- Navigation is not sufficiently intelligent to ignore floaters (at the time of writing).

_INSTALL_STD_MOUSE_TRACKING. If ON, the default methods ClickContent() in ZWindow and ZScroller will use a ZMouseTracker objec.

_DIALOG_EXTENSIONS If you wish to use any of the dialog enhancements such as list boxes, this should be ON. If OFF, it allows your code to be marginally smaller and you do not need to include the extension object sources in your project.

_ZCOMMANDER_DIALOG_AWARE. This sets whether the OpenSubDialog() method exists in ZCommander. Because ZComander is a required class but ZDialog is not, you can set this to OFF if you do not use any dialogs- your code will be smaller and you don't need to include ZDialog sources in your project.

Warning!

The fact that a data member or method is documented in the Class Reference is not an invitation to use or abuse it! The Class Reference describes what all data members and methods actually do, but not when and how to use them beyond a brief description and the occasional handy hint. For your own sanity, try to understand what things are for before you try to use them!

Class Name: CursorUtilities	Based on: -
Type:	Not a Class
Description:	Functions for handling animated cursors and setting the cursor shape

You will rarely have call to use more than one or two of the functions in CursorUtilities. MacZoop supports animated cursors using a VBL task, so animated cursors are set once then forgotten- you do not have to drive them manually while you carry out lengthy processing tasks.

However, for setting the (static) cursor, you are strongly advised to use **SetCursorShape()** since a) it is a single call instead of the usual GetCursor/SetCursor pair, and b) it will automatically use a colour cursor if there is one and implement it in a safe flicker-free way.

ZApplication automatically initialises the standard animated cursors for you.

Functions:

```
void SetWatchCursor();
```

Starts the watch cursor animating. It will continue to animate until you call StopCursorAnimation(), or when the main event loop is reentered, a which point the application calls StopCursorAnimation() anyway. Usually you can just call this once at the start of a lengthy routine then forget about it.

```
void SetBeachBallCursor();
void SetBusyArrowCursor();
```

As above, but animates the “beachball” and “busy arrow” cursors instead.

```
void StopCursorAnimation();
```

Stops any animating cursor from animating, and resets the static cursor to an arrow. You only need to call this if you need to stop the animation before the main event loop is re-entered.

```
Boolean    CursorAnimating();
```

Returns TRUE if there is an animated cursor running, otherwise FALSE. Rarely needed.

```
void SetCursorShape( short cursID );
```

Sets the static cursor. You should always use this instead of GetCursor/SetCursor, because apart from being one call instead of two, it automatically uses a colour cursor if there is one (‘crsr’ resource) and makes sure that this is set safely (it works around a known bug in the toolbox regarding colour cursors). Note that MacZoop does not support animated colour cursors at

present. If you pass 0 to SetCursorShape(), it sets the standard arrow cursor. You can also pass toolbox constants iBeamCursor, crossCursor, etc if you want, or define your own.

```
short GetModifiers();
```

This returns the current state of the keyboard modifier keys (option, command, etc). This is the true current state, not the state at the last event, so it is a handy way to dynamically set the cursor according to the modifiers. The result is formatted exactly the same as the modifiers field in an EventRecord.

Adding your own animated cursors

The watch and beach-ball cursors are set up for you. If you want to have some other animated cursor types, you need to do a little work. You need to define an ‘acur’ resource that points to some ‘CURS’ resources (n.b. you can’t have colour animated cursors at the moment). You then need to call InitAnimatedCursor(), passing the ID of your ‘acur’ resource. This returns a handle that you should store in a global variable (it needs to be global so that the VBL task can find it). To set this cursor animating, you call StartCursorAnimation(), passing the handle and an increment period. A typical period is 4 to 8. However, there are some gotchas. Before calling this, you must check that a) your handle is not NULL- bad karma if it is- Mac dies, and b) that there isn’t a cursor already animating- CursorAnimating() must return FALSE.

All the other functions in CursorUtilities are off limits to your code, on pain of your application’s early death.

n.b. Your cursor animation can be as long as you want- there is no limit to the number of frames in an animation, but beware that every frame allocates a CursHandle. The ‘CURS’ resources should not be marked as purgeable.

Class Name: ZArray	Based on: ZComrade
Type:	Container Class; User
Description:	Generalised storage class- can store arbitrary data in a dynamic array. Every element must be the same size.

This can be used to store any type of data as long as each element is the same size. Though it could be used for objects, it is much more efficient to use ZObjectArray for this purpose- so use ZArray when you need to store identical-sized data that isn't an object.

Class Definition:

```
class ZArray : public ZComrade
{
protected:
     Handle          a;
     unsigned long   blkSize;
     unsigned long   numElements;
     unsigned long   physicalBlks;

public:

     ZArray( unsigned long elementSize = sizeof(Ptr));
     virtual ~ZArray();

// putting stuff in the array

     virtual void    InsertItem( void* item, const long index );
     virtual void    AppendItem( void* item );
     virtual void    SetArrayItem( void* item, const long index );
     virtual void    ConcatenateArray( ZArray* anArray );

// getting stuff out

     virtual void    GetArrayItem( void* item, const long index );
     virtual long    CountItems();
     virtual long    FindIndex( void* item );

// deleting items

     virtual void    DeleteItem( const long index );
     virtual void    DeleteAll();

// moving items

     virtual void    MoveItem( const long curIndex, const long newIndex );
     virtual void    Swap( const long itema, const long itemb );

// grovelling over the items

     virtual void    DoForEach( IteratorProcPtr aProc, const long ref );

// sorting the items

     virtual void    Sort( SortCmpProcPtr compareProc, const long ref );
     virtual void    Sort();
     virtual void    QSort( QSortProcPtr compareProc );
```

```
     virtual short   Compare( void* itema, void* itemb, const long ref );

// inserting items in a sorted list

     virtual long    InsertSortedItem( void* item, SortCmpProcPtr compareProc,
                                       const long ref = 0 );
     virtual long    InsertSortedItem( void* item, const long ref = 0 );

// finding items in a sorted list

     virtual long    BFindIndex( void* item, SortCmpProcPtr compareProc,
                                       const long ref );

// streaming

     virtual void    ReadFromStream( ZStream* aStream );
     virtual void    WriteToStream( ZStream* aStream );

     virtual void    GetDebugInfoString( Str255 s );
     inline    long GetBlockSize() { return blkSize; };

protected:

     virtual void    InsertElement( const long index );
     virtual void    DeleteElement( const long index );
};
```

Data Members

All data members of ZArray are protected. They are accessible to subclasses of ZArray, but not to other objects.

<**a**> is the Mac handle used to store the array's contents
<**blkSize**> is the size in bytes of each element in the array
<**numElements**> is the number of logical items in the array
<**physicalBlks**> is the number of actual blocks in the handle- note that this may be different to <numElements> since ZArray extends the handle several items at a time when needed to improve efficiency.

Methods

```
     ZArray( unsigned long elementSize = sizeof(Ptr));
     virtual ~ZArray();
```

Constructor and destructor.The parameter <elementSize> sets the size in bytes of the items in the array when it is created. This defaults to the size of a pointer (usually 4 bytes).

```
     virtual void    InsertItem( void* item, const long index );
```

Inserts a single item into the array at index position <index>. <Item> is a pointer to the data to be stored.Items stored at indexes greater than <index> are moved up one place.

```
     virtual void    AppendItem( void* item );
```

Appends a single item to the end of the array. <Item> is a pointer to the data to be stored.

```
     virtual void    SetArrayItem( void* item, const long index );
```

Sets the data of the array item at position <index> to the data whose pointer is passed in <item>. The element at that location must already exist.

```
virtual void    ConcatenateArray( ZArray* anArray );
```

Joins another array to this one. The elements of both arrays must bethe same size. The array to join is passed in <anArray>, and is appended top the end of this one.The passed array is not affected by this method- its data is copied to this array.

```
virtual void    GetArrayItem( void* item, const long index );
```

Copies the data at position <index> to a temporary storage area whose pointer is passed in <item>. It is the responsibility of the caller that <item> points to a valid variable or temporary storage area of sufficient size to contain the data.

```
virtual long    CountItems();
```

Returns the number of items in the array. Index values start at 1, and the maximum index value is given by this method.

```
virtual long    FindIndex( void* item );
```

Returns the index number of the item whose data matches that pointed to by <item>. Note that this is a linear search and it performs a byte by byte comparison of the data to determine whether there is a match (in fact it calls the utility function EqualMem()). For both of these reasons, it is inefficient and should be avoided unless there is no alternative.

```
virtual void    DeleteItem( const long index );
```

Removes the data from the array at <index>. Any data with index n[positions greater than this will be moved down one position to occupy the released space.

```
virtual void    DeleteAll();
```

Removes all of the data from the array, emptying it completely. The item count and handle are reset to zero.

```
virtual void    MoveItem( const long curIndex, const long newIndex );
```

Moves the data at <curIndex> to <newIndex>. Other items in the array are moved as necessary to accommodate the movement. The array size and item count is unaffected.

```
virtual void    Swap( const long itema, const long itemb );
```

Swaps the data stored at indexes <itema> and <itemb>. No other items in the array are affected.

```
virtual void    DoForEach( IteratorProcPtr aProc, const long ref );
```

Allows each data item in the array to be passed in order to the Iterator Proc passed. Note that it is often just as easy to use a simple for{...} loop. The Iterator Proc takes the form:

```
typedef void (*IteratorProcPtr)( void* item, const long ref );
```

A pointer to the data ispassed in <item>, and the value passed to DoForEach() in the <ref> parameter is passed in <ref>. You can use this parameter for any desired purpose.

```
virtual void    Sort( SortCmpProcPtr compareProc, const long ref );
```

Sorts the data in the array into order. The order is defined using the <compareProc>, which is a procedure of type SortCmpProcPtr. This procedure examines two items of data and returns a value indicating the relative ordering of the items. The Sort() method uses a Shellsort algorithm and is extremely fast. The comparison proc is of the form:

```
typedef short (*SortCmpProcPtr)( void* itema, void* itemb, const long ref );
```

Pointers to two items to compare are passed in <itema> and <itemb>. The function must compare the data as it sees fit and return:

- -1 if a is earlier in the list than b
- 0 if and b are equal
- 1 if a is later in the list than b

(n.b. to sort in reverse order, simply return the inverse result).

The macro CMP is useful for comparing numerical data within this function. For string data, the toolbox function RelString() is often useful.

The parameter <ref> passed to the original Sort method is passed to the comparison proc in <ref>. You can use this parameter for any desired purpose.

```
virtual void    Sort();
```

This is a variation on the Sort method which uses the Compare() method instead of a comparison function. To use this method for sorting, you have to subclass ZArray.

```
virtual void    QSort( QSortProcPtr compareProc );
```

Another sort function. This uses the ANSI qsort function to implement the sort algorithm. This can sometimes be even faster than the Shellsort method. However, to use this method, you need to link to the StdCLib, and you can’t use a ref p[arameter as you can with internal sorting. The comparison function for QSort takes the form:

```
typedef int       (*QSortProcPtr)( const void* itema, const void* itemb );
```

Pointers to two data items to compare are passed in <itema> and <itemb>, the function should return as for the standard Sort comparison function.

```
virtual short   Compare( void* itema, void* itemb, const long ref );
```

If you use the parameterless version of Sort(), you should overrid ethis method to implement your comparison function. Not required if you pass a comparison function explicitly to Sort().

```
virtual long    InsertSortedItem( void* item, SortCmpProcPtr compareProc,
                                  const long ref = 0 );
```

If you are using sorted arrays, it is desirable to keep it sorted as much as possible. For efficiency therefore, You can insert data into the array in the correct position using this method. The same comparison function must be passed to this method that you use for Sort(). The array must be initially sorted or else empty for this to work correctly. The index position where the item was inserted is returned.

```
virtual long    InsertSortedItem( void* item, const long ref = 0 );
```

This version of the method should only be used if you have subclassed ZArray and overridden the Compare() method. It does the same as the above, but relies on the overridden comparison method rather than an explicit comparison function.

```
virtual long    BFindIndex( void* item, SortCmpProcPtr compareProc,
                                    const long ref );
```

This method performs a binary search on a sorted array. It uses the same comparison function used for sorting, and this is used to compare items while doing a binary search in oredr to locate the item's position. It returns the position of the found data, or else the index of the nearest position (the index where the item would be inserted if it subsequently was passed to InsertSortedItem). This is a very fast and efficient search, but requires that the array is sorted.

```
virtual void    ReadFromStream( ZStream* aStream );
virtual void    WriteToStream( ZStream* aStream );
```

Reads and Writes the array to the given stream. All of the data in the array is streamed.

```
virtual void    GetDebugInfoString( Str255 s );
```

Returns some readable information that can be displayed by runtime inspectors and debuggers.

```
inline    long GetBlockSize() { return blkSize; };
```

Returns the size in bytes of each item in the array.

```
virtual void    InsertElement( const long index );
virtual void    DeleteElement( const long index );
```

Protected methods which manipulate the handle when storing and removing data. The index values passed here are 0-based, not 1-based as elsewhere in ZArray.

ZArray Messages

ZArray uses comrade messaging. By listening to an array, you can monitor its activities. ZArray employs the following messages:

```
enum
{
     msgArrayItemAdded = 'arr1',
     msgArrayItemDeleted,
     msgArrayItemMoved,
     msgArrayItemChanged,
     msgArrayItemInserted,
     msgArrayAllDeleted
};
```

Class Name: ZApplication	Based on: ZCommander
Type:	Framework Class; User
Description:	Implements the general Macintosh Application Programming Model. Usually subclassed.

ZApplication implements the general Macintosh Programming model. There is one and only one instance of this class in an application, and the global variable gApplication must point to it. ZApplication is often subclassed.

Class Definition

```
class ZApplication : public ZCommander
{
      friend class ZEventHandler;

protected:

      Boolean         done;           // normally FALSE, if set TRUE, will try to quit
      short           phase;          // current phase
      short           appResRefNum;   // refnum of application resource file
      ZEventHandler*  zEH;            // event handler object
      Handle          shortageFund;   // to deal with tight memory, we can release this
      FTypeListHdl    itsFileTypes;   // list of filetypes we can open
      ZUndoTask*      curUndoTask;    // current undoable task
      ZPrinter*       itsPrinter;     // printer object for handling print commands
      Boolean         memIsShort;     // flag memory problem
      Boolean         userHasSeenAlert;// TRUE if user has been warned about the memory
      Boolean         splashVisible;  // TRUE if start-up splash is displayed
      short           msDepth;        // screen depth of main screen
      short           fFrontSleep;    // front sleep value
      short           fBackSleep;     // background sleep value

public:

      ZApplication();
      virtual ~ZApplication();

      // initialisation and clean-up

      virtual void              InitMacZoop( const short numMasterBlocks = 8 );
      virtual void              StartUp() {};
      virtual void              ShutDown() {};
      virtual void              ReadPrefs() {};
      virtual void              ShowSplash() {};
      virtual void              RunFirstTask() {};

      // event processing

      virtual void              Run();
      virtual Boolean           Quit();
      virtual void              RequestQuit();
      virtual Boolean           MemoryShortage( const Size bytesShort );
      virtual void              Process1Event( const short mask = everyEvent );
      virtual void              Process1Event( EventRecord* anExternalEvent );
      virtual void              ProcessAllEvents();
      virtual void              HandleCommand( const long aCmd );
```

```
      virtual void              HandleCommand( const short menuID, const short itemID
);
      virtual void              UpdateMenus();
      virtual Boolean           GetCurrentEvent( EventRecord* anEvent );
      virtual void              HandleAppleEvent(    AEEventClass aeClass,
                                                     AEEventID aeID,
                                                     AppleEvent* aeEvt,
                                                     AppleEvent* reply );

      virtual void              MouseNotInAnyWindow( const Point globalMouse );
      virtual void              HandleMBarHiding( const Point globalPt );
      virtual void              WaitApplicationForeground();
      virtual void              ProcessHLEvent( const EventRecord& theEvent ) {};

      // status utilities

      virtual    short          GetClicks();
      virtual    Boolean        InBackground();
      virtual void              GetName( Str255 appName );

      virtual void              DoSuspend();
      virtual void              DoResume();

      // error processing

      virtual void              HandleError( OSErr theErr );

      // window construction

      virtual ZWindow*          OpenNewWindowType( OSType aType = 0 );
      virtual void              CloseAll( Boolean closeFloaters = FALSE );
      virtual ZWindow*          GetFrontWindow();
      virtual void              AboutBox();
      virtual void              DoPreferences() {};
      virtual void              ProcessMenuTearOff( const short menuID,
                                                    const Point tearOffDropLoc ){};

      // opening files

      virtual Boolean           PickFile( FSSpec* aFile, OSType* fType );
      virtual Boolean           PickFile( FSSpec* aFile, OSType* fType,
                                          FTypeListHdl fTypesList );
      virtual ZWindow*          OpenFile( const FSSpec& aFile, const OSType fType,
                                          Boolean isStationery = FALSE );

      // extending the file types

      virtual void              AddFileType( const OSType aType );
      virtual Boolean           CanOpenFileType( const OSType aType );

      // undo task handling

      virtual void              SetTask( ZUndoTask* aTask );
      virtual void              UpdateUndo();
      inline ZUndoTask*         GetUndoTask() { return curUndoTask; };

      // printer handling

      virtual void              MakePrinter();
      virtual void              DoPageSetup();
      virtual void              DoPrint();
      inline    ZPrinter*       GetPrinter() { return itsPrinter; };

      // other inline accessors

      inline    short           GetPhase()                          { return phase;};
```

```
      inline    FTypeListHdl    GetFileTypeList()    { return itsFileTypes; };
      inline    Boolean         MemoryCrisis()       { return memIsShort; };
      inline    Boolean         UserHasSeenMemoryCrisisAlert()  { return
                                                                  userHasSeenAlert; };
      inline    ZEventHandler*  GetEventHandler()    { return zEH; };
      inline    short           GetAppRefnum()       { return appResRefNum; };

      inline    void            SetFrontSleep( short aSleep )  { fFrontSleep = aSleep;
};
      inline    void            SetBackSleep( short aSleep )   { fBackSleep = aSleep;
};

      virtual void              GetDebugInfoString( Str255 s );

      // process stuff:

      virtual void              GetProcessSerialNumber( ProcessSerialNumber* PSN );
      virtual void              GetProcessInfo( ProcessInfoRec* pInfo );
      virtual void              GetProcessLocation( short* volume, long* parentDirID );

protected:

      void                      InitMacApplication( const short numMasterBlocks );
      virtual void              MakeHelpers();
      virtual void              MakeClipboard();
      virtual Boolean           CheckCanRun();
      virtual void              InitMenuBar();
      virtual void              CheckLowMemory();
      virtual void              RegisterClasses();

      virtual ZWindow*          MakeNewWindowType( OSType aType = 0 );
};
```

Data Members

<**done**> Normally FALSE, the application will quit if set to TRUE (see RequestQuit();)
<**phase**>, the current phase, Can be one of:

- kInitialising
- kRunning
- kQutting

<**appResRefNum**> isthe file referncenumber of the application’s resource fork.
<**zEH**> is the reference to the ZEventHandler object being used by the application
<**shortageFund**> is the handle to the reserve memory
<**itsFileTypes**> is the internal list of filetypes displayed by the PickFile() method
<**curUndoTask**> is the object reference to the currenr undo task, if any.
<**itsPrinter**> isthe object reference to the printer object
<**memIsShort**> is TRUE if part of the shortage fund has been released
<**userHasSeenAlert**> is TRUE oncethe user has seen the shortage warning alert once. This stops it being shown again until the crisis has been resolved.
<**splashVisible**> is TRUE if the splash screen has been displayed. ZApplication does not create a splash screen, but your own application might.
<**msDepth**> is the pixel depth of the main monitor.
<**fFrontSleep**> is the value of slep used when in the foreground
<**fBackSleep**> is the value of sleep used when the application is in the background.

Methods

```
ZApplication();
virtual ~ZApplication();
```

Constructor and destructor. The constructor initialises the application's data members, the gMacInfo global and the cursors. The destructor performs some tidying up- much more if _SLOW_BUT_SURE_DESTRUCTION is set to ON.

```
virtual void              InitMacZoop( const short numMasterBlocks = 8 );
```

Initialises the Mac toolbox and everything else needed by MacZoop. It registers streaming classes, sets up the shortage fund, creates the helper objects, sets up the printer, clipboard and menu bar and registers the application with Appearance and Navigation Services if required.

```
virtual void              StartUp() {};
```

You can override this method to perform your own additional initialisation when your application start up. It is called after most ofthe standard initialisation has bene done.

```
virtual void              ShutDown() {};
```

You can overridethis to perform additional work when the application quits. By the time this is called, the application is committed to actually quitting.

```
virtual void              ReadPrefs() {};
```

You can override this method to read your preferences file as required. It is called during initialisation, but somewhat earlier than StartUp(). The menu bar and printer do not yet exist when this is called. Notethat the prefs file is not automatically created or opened- you should do that in this method. (see ZPrefsFile)

```
virtual void              ShowSplash() {};
```

You can override this to create a splash window if required.This is called during initialisation, after the prefs is read, but before menu bar and printer are created. It is OK to use ZWindow to create a splash screen if you wish. You should remove any splash screen you display with you RunFirstTask() method.

```
virtual void              RunFirstTask() {};
```

Overridethis method to perform start-up tasks under the protection of the standard exception and error handling. Exceptions that arise during StartUp() or other initialisation method will cause a fatal error and your application will not launch. Generally, your fixed or global windows should be created in the RunFirstTask() method, and also any splash screen you displayed should be removed here.

```
virtual void              Run();
```

The main event processing method. You should not override or call this directly.

```
virtual Boolean           Quit();
```

The main quit processing method. This closes all open windows by default. Rarely should you override this- use ShutDown() instead.

```
virtual void              RequestQuit();
```

This politely tells the application to Quit. It will do so when able, but bear in mind that th euser may abort the quit, so you are not guaranteed to quit when this is called.

```
virtual Boolean           MemoryShortage( const Size bytesShort );
```

Called by the Mac memory managerwhen it is unable to fulfill a memory request. The default method releases some or all of the shortage fund to try to make the memory available. You can override this to take further steps as needed.

```
virtual void              Process1Event( const short mask = everyEvent );
virtual void              Process1Event( EventRecord* anExternalEvent );
```

Fetches and dispatches one event. The first version of the method uses ZEventHandler's GetAnEvent() method to fetch an event from the queue. The second version accepts an event already fetched. This is sometimes done by libraries, e.g. Navigation Services. Process1Event is called by Run() and other methods as needed.

```
virtual void              ProcessAllEvents();
```

This fetches and dispatches all events until a null event is returned, at which point it returns to the caller. This can be used to "mop up" a series of pending events in one go. Rarely needed.

```
virtual void         HandleCommand( const long aCmd );
virtual void         HandleCommand( const short menuID, const short itemID );
```

Handles the standard menu commands About, New, Open, Page Setup, Print, Undo, Quit, Preferences, Stack and Tile Windows. You can override this to handle additional commands, calling the inherited method to handle the standard ones. The other version of the method handles the desk accessory titles in the Apple menu.

```
virtual void              UpdateMenus();
```

Enables the menu commands it can handle.

```
virtual Boolean           GetCurrentEvent( EventRecord* anEvent );
```

Returns the event that it is currently processing.This may be useful if you want to obtain event data that was not passed down during dispatch. This is rare.

```
virtual void              HandleAppleEvent(    AEEventClass aeClass,
                                               AEEventID aeID,
                                               AppleEvent* aeEvt,
                                               AppleEvent* reply );
```

Handles the four required apple events- Open Aplication, Open Documents, Print Documents and Quit. You can override this to handle other apple events that you registered.

```
virtual void              MouseNotInAnyWindow( const Point globalMouse );
```

Called when the mouse is not in any window that belongs to the application. The default method rests the cursor to an arrow.

```
virtual void              HandleMBarHiding( const Point globalPt );
```

Called when needed to handle automatic menu bar hiding. This can be enabled using the _AUTO_MBAR_HIDING ProjectSetting. You should not call or override this, in general.

```
virtual void              WaitApplicationForeground();
```

Waits for the application to be brought to the front, when it returns to the caller. It returns immediately if the application is not suspended.

```
virtual void              ProcessHLEvent( const EventRecord& theEvent ) {};
```

You can overridethis method to process High Level (HL) events of your own devising. Th edefault method does nothing- standard Apple Events and suspend, resume, etc are handled elsewhere.

```
virtual    short          GetClicks();
```

Returns the number of mouse clicks in a particular place. It returns 1 for a single-click, 2 for a double, 3 for triple, etc. Clicks must all be within the double click period set by the system and in the same spatial location as defined by the window that was clicked (see handling double-clicks).

```
virtual    Boolean        InBackground();
```

Returns TRUE if the application is suspended.

```
virtual void              GetName( Str255 appName );
```

Retuns the name of the application as a pascal string.

```
virtual void              DoSuspend();
virtual void              DoResume();
```

The default methods pass on the suspend and resume events to the window manager and perform some cursor management. You can override these to do further processing, but be sure to call the inherited methods.

```
virtual void              HandleError( OSErr theErr );
```

Handles standard error user-interface by displaying the error alert (usingthe notification manager if the application is suspended). You can override this if you want to handle errors in a different way.

```
virtual ZWindow*          OpenNewWindowType( OSType aType = 0 );
```

Handles the standard New command by creating an appropriate window object for the default file type passed (see MakeNewWindowType).

```
virtual void              CloseAll( Boolean closeFloaters = FALSE );
```

Closes all open windows, checking their save status as needed. This is called when the application quits, but can also be called by option-clicking the close box of a window or choosing option-Close in the File menu. Floaters are not normally closed by this unless you pass TRUE.

```
virtual ZWindow*          GetFrontWindow();
```

Returns the frontmost non-floating window object.

```
virtual void              AboutBox();
```

Displays the about box. By default, this is an alert with ID 128. You can override this method to set up a more elaborate about box, or elsesimply modify the resources.

```
virtual void              DoPreferences() {};
```

Called in response to the standard Preferences command. Override this to display your preferences dialog box, etc.

```
virtual void              ProcessMenuTearOff( const short menuID,
                                              const Point tearOffDropLoc ){};
```

Called when a supporting menu is “torn off”. You can override this to implement tear-off menus as required. This requires a number of steps to implement a sensible tear-off scheme.

```
virtual Boolean           PickFile( FSSpec* aFile, OSType* fType );
virtual Boolean           PickFile( FSSpec* aFile, OSType* fType,
                                    FTypeListHdl fTypesList );
```

Displays the Standard File or Navigation dialog for the user to select a file to open. If a file is picked, the method returns TRUE and <aFile> and <fType> are the spec and type of the file picked. The internal file types list is used to set what files are visible in the dialog. In the second version of this method, you can pass in a file types list of your own. To create such a list, you can use the utlity function NewFileTypesList() (ZoopUtilities.cpp). The reason for the opaque file types list is to provide the same interface whether or not we have Navigation Services available.

```
virtual ZWindow*          OpenFile( const FSSpec& aFile, const OSType fType,
                                    Boolean isStationery = FALSE );
```

Opens the file passed in <aFile> by creating the required window object and passing it the file information. The window opens the file then this places and selects it. Called as part of standard processing of the Open command and “open documents” apple event, etc.

```
virtual void              AddFileType( const OSType aType );
```

Adds a given file type to the internal list of file types. This allows you to set up the list programatically.

```
virtual Boolean           CanOpenFileType( const OSType aType );
```

Returns TRUE if the file type is in the internal list of file types.

```
virtual void              SetTask( ZUndoTask* aTask );
```

Sets a given undo task as the current task, discarding the old one, if any. You can overridethis to do different Undo processing, for example multiple undos.

```
virtual void              UpdateUndo();
```

Updates the Undo menu command. Called as needed- you should probably override this if you have multiple undos.

```
inline ZUndoTask*         GetUndoTask();
```

Returns the current Undo task.

```
virtual void              MakePrinter();
```

Creates the ZPrinter helper object and sets it up. This object is responsible for handling the Print command, etc.

```
virtual void              DoPageSetup();
virtual void              DoPrint();
```

Responds to the standard commands Page Setup and Print. The default methods simply call the ZPrinter object to do the dirty work.

```
ZPrinter*       GetPrinter();
```

Returns the printer object.

```
short           GetPhase();
```

Returns the current runtime phase of the application.

```
FTypeListHdl    GetFileTypeList();
```

Returns the internal file types list. The format is private and subject to change, so use with caution.

```
Boolean         MemoryCrisis();
Boolean         UserHasSeenMemoryCrisisAlert();
```

Returns information about the memory status. MemoryCrisis() returns TRUE if part of the shortage fund has been released but not replenished. UserHasSeenMemoryCrisisAlert() returns TRUE if the user has been told.

```
ZEventHandler*  GetEventHandler();
```

Retuns the ZEventHandler object.

```
short           GetAppRefnum();
```

Retuns the file reference number of the application’s resource fork.

```
void            SetFrontSleep( short aSleep );
void            SetBackSleep( short aSleep );
```

Sets the sleep time for when the application is in the foreground. Similarly, SetBackSleep sets the sleep time for when the application is in the background.

```
virtual void              GetDebugInfoString( Str255 s );
```

Returns readable information that can be displayed by runtime inspectors and debuggers.

```
virtual void              GetProcessSerialNumber( ProcessSerialNumber* PSN );
```

Returns the MacOS process serial number forthe application

```
virtual void              GetProcessInfo( ProcessInfoRec* pInfo );
```

reuns the MacOS Process Info for the application. The caller must allocate sufficient space for the result- consult Inside Macintosh.

```
virtual void              GetProcessLocation( short* volume, long* parentDirID );
```

Returns the volume and directory ID of the folder containing the application on disk. This is useful when creating or locating files and folders in the same folder as the application.

```
void                      InitMacApplication( const short numMasterBlocks );
```

Initialises the toolbox at startup. Also sets up master pointers and the internal file types list based on the application's BNDL resources.

```
virtual void              MakeHelpers();
```

Creates the various helper objects. You can overridethis to make other objects or to replace the helpers with objects of your own.

```
virtual void              MakeClipboard();
```

Creates the clipboard object. Override this if you have your own clipboard object. (Must be a subclass of ZClipboard).

```
virtual Boolean           CheckCanRun();
```

Returns TRUE if we have System 7.0 or later. You can overridethis to perform other checks about the suitability of the host machine to run your application. Returning FALSE causes an apologetic message to be displayed and the application will not launch.

```
virtual void              InitMenuBar();
```

Initialises the menubar, building it and the command structure using the application's MBAR and associated resources. Note that by default, MBAR ID 128 is used to set up ZMenuBar. Normally, you would just edit this resource as you need, but you could override this method to use a different resource if you needed to.

```
virtual void              CheckLowMemory();
```

Handles the checking of the shortage fund and display of the alert to the user. You could override this if you wanted some other behaviour, but this would be very unusual.

```
virtual void              RegisterClasses();
```

Registers the standard streamable classes, if streaming is turned on. If you have your own streamable objects, you should override this and register them, and call the inherited method to register the standard classes.

```
virtual ZWindow*          MakeNewWindowType( OSType aType = 0 );
```

Creates and initialises a ZWindow or derivative object suitable to handle the file type passed. This is commonly overridden, especially if you have more than one window or file type in your application. The standard method makes a window of whatever type is set up in the ProjectSettings.h file. If you need more than one type, you must override this. Create the window according to the file type, call its InitZWindow method and return it. This method is called by a number of other functions with ZApplication.

ZApplication Messages

```
msgApplicationSuspending      = 'susp',
msgApplicationResuming        = 'rsum',
msgMainScreenDepthChanged     = 'msdx'
```

Class Name: ZClipboard	Based on: ZComrade
Type:	Framework Class
Description:	Provides object-oriented API to Macintosh clipboard (desk scrap). Can be subclassed for private scrap schemes.

ZClipboard manages the standard Macintosh global scrap, and provides a simple API to it. It can be subclassed to provide private scrap schemes, etc. There should be only one instance of ZClipboard, and the global <gClipboard> is a reference to it.

Class Definition

```
class ZClipboard : public ZComrade
{
public:
     ZClipboard() : ZComrade() { classID = CLASS_ZClipboard; gClipboard = this; };
     ~ZClipboard() {};

// putting data on the clipboard
     virtual void    PutData( OSType dataType, Handle someData );
     virtual void    PutData( OSType dataType, Ptr dataPtr, const long dataLen );
     virtual void    PutData( PicHandle aPicture );
     virtual void    PutText( Handle textH );
     virtual void    PutText( Ptr charBuf, const long textLen );

     virtual void    AppendData( OSType dataType, Handle someData );
     virtual void    AppendData( OSType dataType, Ptr dataPtr, const long dataLen );
     virtual void    AppendData( PicHandle aPicture );
     virtual void    AppendText( Handle textH );
     virtual void    AppendText( Ptr charBuf, const long textLen );

// clearing the clipboard
     virtual void    Clear();

// getting data and info
     virtual Handle  GetData( OSType dataType );
     virtual Boolean QueryType( OSType dataType );
     virtual long    GetDataSize( OSType dataType );
     virtual short   GetClipStatus();

// private scrap conversion
     virtual void    ConvertFromPrivate() {};
     virtual void    ConvertToPrivate() {};

// multi-data handling stuff
     virtual short   CountTypes();
     virtual OSType  GetIndType( const short index );

protected:
     Boolean         GetWildcardType( OSType* aType );
};
```

Data Members

none.

Methods

```
virtual void    PutData( OSType dataType, Handle someData );
virtual void    PutData( OSType dataType, Ptr dataPtr, const long dataLen );
virtual void    PutData( PicHandle aPicture );
```

PutData places the data on the clipboard, replacing anything that is there already. The various varieties of the method allow you to pass a Handle with data type, arbitrary data with a length and data type, and a Picture handle, which is placed as type ‘PICT’.

```
virtual void    PutText( Handle textH );
virtual void    PutText( Ptr charBuf, const long textLen );
```

Similar to PutData, it replaces the current contents of the clipboard with the text passed as a Handle or character buffer. Data type is ‘TEXT’.

```
virtual void    AppendData( OSType dataType, Handle someData );
virtual void    AppendData( OSType dataType, Ptr dataPtr, const long dataLen );
virtual void    AppendData( PicHandle aPicture );
```

AppendData adds the given data to the clipboard, but does not replace the current contents. You may recall that the clipboard can contain several items of data, differently typed, and the Paste target may select the most appropriate one. This method is used when you want to add mor ethan one item of data to the clipboard.

```
virtual void    AppendText( Handle textH );
virtual void    AppendText( Ptr charBuf, const long textLen );
```

As above, but for ‘TEXT’ data.

```
virtual void    Clear();
```

Empties the clipboard. This is called by PutData.

```
virtual Handle  GetData( OSType dataType );
```

Returns a new handle to the data of the given type, or NULL if no such data exists. The caller should dispose of the handle after using it. You can also pass <Wild_Card> in dataType to return data whatever its type.

```
virtual Boolean QueryType( OSType dataType );
```

Returns TRUE if the given type of data exists on the clipboard.You can use this method when checking whether to enable the Paste command for any particular target.

```
virtual long    GetDataSize( OSType dataType );
```

Returns the size in bytes of the data of the given type on the clipboard, or -1 if the type does not exist.

```
virtual short   GetClipStatus();
```

Retuns the status value of the standard desk scrap. This is just a number that will change if the contents of the clipboard have changed. By comparing it to a previously obtained count, you can see if e.g. a clipboard window needs updating.

```
virtual void    ConvertFromPrivate() {};
virtual void    ConvertToPrivate() {};
```

These two methods are called when the application suspends and resumes, respectively. Th edefault methods do nothing, but if you have subclass ZClipboard to implement a private scrap scheme, you should override these methods so that you can convert your private clip data to public clip data and vice versa.

```
virtual short   CountTypes();
```

This returns a count of the number of different data types on the clipboard.

```
virtual OSType  GetIndType( const short index );
```

This returns the actual data type of the indexed data. <index> can range from 1 to the value returned from CountTypes(). You can then use GetData as normal to obtain the data.

```
Boolean         GetWildcardType( OSType* aType );
```

Returns the type in <aType> of the data on the clipboard (or first item if more than one). If there is no data at all, returns FALSE, otherwise TRUE. This is one step of obtaining so-called "wildcard" data- that is, get the data whatever type it is. This is used only in special circumstances, since we can rarely paste data of any arbitrary type. One exception would be e.g. the hex editor, which accepts data in any form.

ZClipboard Messages

```
enum
{
     clipContentsChanged = 'clp1',
     clipContentsAppended,
     clipContentsCleared,
     clipDataConverted
};
```

Class Name: ZCommander	Based on: ZComrade
Type:	User Class; Abstract
Description:	Implements link in the command chain

ZCommander is the basis for all objects in MacZop that can respond to commands. This includes all windows, the application, and dialog items. Commanders can also have attached ZModifier objects, and also ZTimer.

Class Definition

```
class ZCommander : public ZComrade
{
protected:
      ZCommander*          itsBoss;
      ZCommanderList*      itsUnderlings;
      ZModifierList*       itsModifiers;

public:
      ZCommander( ZCommander* aBoss );
      ZCommander();
      virtual ~ZCommander();

      virtual void    HandleCommand( const short menuID, const short itemID);
      virtual void    HandleCommand( const long theCommand );
      virtual void    UpdateMenus();
      virtual void    HandleAppleEvent(    AEEventClass aeClass,
                                           AEEventID aeID,
                                           AppleEvent* aeEvt,
                                           AppleEvent* reply );

      virtual void    Idle();
      virtual void    Type( const char theKey, const short modifiers );
      virtual void    SendMessage( long aMessage, void* msgData );
      virtual void    SendMessage( ZMessage* aMessage );
      virtual void    DoTimer( long timerID ) {};

      virtual void    DoSuspend();
      virtual void    DoResume();

      virtual Boolean GetBalloonHelp( const Point mouseIn,
                                      Rect* rectOut,
                                      Point* tipOut,
                                      HMMessageRecord* hmOut ) { return FALSE; };

      virtual void    DoCut() { DoCopy(); DoClear(); };
      virtual void    DoCopy() {};
      virtual void    DoPaste() {};
      virtual void    DoClear() {};
      virtual void    DoSelectAll() {};
      virtual Boolean CanPasteType() { return FALSE; };

      virtual ZDialog*     OpenSubDialog( const short dlogID );

      virtual void    ContextualMenuClick( Point globalMouse ) {};
```

```
     virtual    ZCommander*     GetHandler() { return this; };
     inline     ZCommander*     GetBoss(){ return itsBoss; };
     inline     ZCommanderList* GetUnderlings() { return itsUnderlings; };

// streaming:

     virtual void    WriteToStream( ZStream* aStream );
     virtual void    ReadFromStream( ZStream* aStream );

// modifiers a.k.a. "attachments"

     virtual void    AddModifier( ZModifier* aModifier );
     virtual void    RemoveModifier( ZModifier* aModifier );
     virtual Boolean ExecuteModifiers( const long modifierMessage, void* modData );

protected:

     virtual void    AddUnderling( ZCommander* anUnderling );
     virtual void    RemoveUnderling( ZCommander* anUnderling );
};
```

Data Members

<**itsBoss**> is the object that is the boss of this one, that is, the next link up the command chain from this object.
<**itsUnderlings**> is the list of obejcts that this is the boss of.
<**itsModifiers**>is the list of ZModifier objects attached to this commander.

Methods

```
     ZCommander( ZCommander* aBoss );
     ZCommander();
     virtual ~ZCommander();
```

Constructors and destructor. The boss of the object is passed in the constructor, and must be a valid ZCommander object. The default constructor is used only for creating commander objects from a stream.

```
     virtual void    HandleCommand( const short menuID, const short itemID);
     virtual void    HandleCommand( const long theCommand );
```

Override these methods to intercept and handle commands. Th edefault methods disptach the commands for Cut, Copy, Paste, Select All and Clear, and pass all others to the boss.

```
     virtual void    UpdateMenus();
```

Override this method to enable menu commands that you can handle, according to the state variables of your object. You should also use this method to set the text of menu commands, and any check marks, etc. The default method passes on thecall to the boss.

```
     virtual void    HandleAppleEvent(    AEEventClass aeClass,
                                          AEEventID aeID,
                                          AppleEvent* aeEvt,
                                          AppleEvent* reply );
```

Override this method to handle apple events destined for this object. Th edefault method passes on the call to the boss.

```
virtual void    Idle();
```

This is called repeatedly as long as your commander is part of the current command chain. You can use it to perform idle processing, etc. While useful, it may be more appropriate to use a timer to get periodic time, since Idle() is not called for inactive commanders. The default method passes the call on to the boss.

```
virtual void    Type( const char theKey, const short modifiers );
```

Handle keyboard input to the commander. Overridethis method to respond to the keyboard when your commander is active. Inactive commanders will not receive this message. The default method passes the message to the boss.

```
virtual void    SendMessage( long aMessage, void* msgData );
virtual void    SendMessage( ZMessage* aMessage );
```

Overrides ZComrade so that the commander's boss always gets notified of any messages sent by the commander without an explicit link being set up.

```
virtual void    DoTimer( long timerID ) {};
```

Override this method to respond to timer calls that you have set up. The ID number of the timer is passed so you can identify which timer is calling if you set up more than one. This method is called even when your commander is not part ofthe active command chain.

```
virtual void    DoSuspend();
virtual void    DoResume();
```

Called when the application is suspended and resumed, when your commander is part of the command chain. Override these to respond to this message.

```
virtual Boolean GetBalloonHelp( const Point mouseIn,
                                Rect* rectOut,
                                Point* tipOut,
                                HMMessageRecord* hmOut );
```

Override this to construct a balloon help message for the commander. This is usually only doneif the commander is a real user-interface object, such as a dialog item or window.

```
virtual void    DoCut();
virtual void    DoCopy();
virtual void    DoPaste();
virtual void    DoClear();
virtual void    DoSelectAll();
```

Standard methods for handling the associated commands. You can overridethese as needed to implement the comands. ZCommander already dispatches these commands so you don't have to look for them in HandleCommand(). Notethat DoCut() calls {DoCopy(); DoClear();} which is usually adequate.

```
virtual Boolean CanPasteType();
```

Overridethis to help manage the Paste command. You should query the clipboard and return TRUE if there is data there you are able to paste,otherwise FALSE.

```
virtual ZDialog*     OpenSubDialog( const short dlogID );
```

Creates and displays the dialog with the resource ID pased as an underling of this commander. This is appropriate if you are not subclassing ZDialog, and can handle the dialog either inline or by responding to its messages. Returns the dialog object created.

```
virtual void    ContextualMenuClick( Point globalMouse ) {};
```

Overridethis if you want to display a contextual menu for this commander. This is called if the user control-clicks your commander.

```
virtual    ZCommander*     GetHandler() { return this; };
```

Returns the commander that begins the command chain relative to this one.Normally, this is the object itself, but if you have a window that contains other commanders that can begin the command chain, you should override to return the one that has the current focus. The event handler is responsible for setting up the command chain, but only works down to the level of the window. This function is needed to more finely resolve the command chain if the window contains other commanders.

```
ZCommander*     GetBoss();
```

Returns the commander’s boss.

```
ZCommanderList* GetUnderlings();
```

Returns the commanders list of underlings. You should generally not use this.

```
virtual void    WriteToStream( ZStream* aStream );
virtual void    ReadFromStream( ZStream* aStream );
```

Reads and Writes the commander to a stream.

```
virtual void    AddModifier( ZModifier* aModifier );
```

Attaches a modifiers (an object of type ZModifier) to this commander. A Modifier is an object that is called when certain things occur and can modify or change the event. (see ZModifier)

```
virtual void    RemoveModifier( ZModifier* aModifier );
```

Removes the modifier from the commander.

```
virtual Boolean ExecuteModifiers( const long modifierMessage, void* modData );
```

Passes the message to all of the attached modifiers, along with any data for the message. The modifier can respond to the message or modify it. If it returns TRUE, processing continues. If FALSE, the modifier completely handled the message and does not want the standard processing to occur.

```
virtual void    AddUnderling( ZCommander* anUnderling );
virtual void    RemoveUnderling( ZCommander* anUnderling );
```

Internal methods that manage the command chain linking.

Class Name: ZComrade		Based on: -
Type:	Abstract Class; User	
Description:	Abstract inter-object messaging protocol. A very useful base class for many purposes.	

ZComrade is used to pass messages between objects in a very general way.

Class Definition

```
class ZComrade  : public ZObject
{
     private:
          ZComradeList*        talkers;
          ZComradeList*        listeners;

     public:
          ZComrade();
          virtual ~ZComrade();

          virtual void    SendMessage( long aMessage, void* msgData );
          virtual void    SendMessage( ZMessage* aMessage );
          virtual void    ReceiveMessage( ZComrade* aSender,
                                          long theMessage, void* msgData ) {};
          virtual void    ReceiveMessage( ZComrade* aSender, ZMessage* aMessage ) {};
          virtual void    ListenTo( ZComrade* aSender );
          virtual void    StopListeningTo( ZComrade* aSender );

     // streaming:

          virtual void    WriteToStream( ZStream* aStream );
          virtual void    ReadFromStream( ZStream* aStream );

     protected:

          virtual void    AddTalker( ZComrade* aTalker );
          virtual void    AddListener( ZComrade* aListener );
          virtual void    RemoveTalker( ZComrade* aTalker );
          virtual void    RemoveListener( ZComrade* aListener );
};
```

Data Members

<**talkers**> the list of objects this is listening to
<**listeners**> the list of objects listening to this

Methods

```
          virtual void    SendMessage( long aMessage, void* msgData );
          virtual void    SendMessage( ZMessage* aMessage );
```

Sends the message to all of the listeners, along with any data passed along with the message. The messag ecould also be implemented as an object of type ZMessage, which implementation is left up to you. Usually the first form is used.

```
virtual void    ReceiveMessage( ZComrade* aSender,
                                long theMessage, void* msgData ) {};
virtual void    ReceiveMessage( ZComrade* aSender, ZMessage* aMessage ) {};
```

Overridethis method to respond to messages sent to you. <aSender> is the object sending the message. If you prefer to implement messages as objects, you can use the second form.

```
virtual void    ListenTo( ZComrade* aSender );
```

Establishes a communication link between two ZComrade objects. The object that ‘owns” this method is the one that becomes the listener of the other. If you require two way communication, call this twice, once for each object.

```
virtual void    StopListeningTo( ZComrade* aSender );
```

Breaks a previously established link.

```
virtual void    WriteToStream( ZStream* aStream );
virtual void    ReadFromStream( ZStream* aStream );
```

Reads and Writes the ZComrade object to the stream.

The other protected methods are all internal link management methods and should not be used directly.

Class Name: ZErrors		Based on: -
Type:	Not A Class	
Description:	Handy error condition detectors and exception throwers.	

See Also: Error Handling

Functions

```
void FailNIL( void* aPtr );
```

FailNIL() tests any pointer or Handle or Object Reference to see if it is NULL. If so, an exception <memFullErr> (-108) is thrown. Otherwise does nothing.

```
void FailOSErr( OSErr theErr );
```

FailOSErr() tests an error code passed to it to see if it is <noErr> (0), if not, an exception of that code is thrown. This is useful to enclose functions that return OSErr error codes as their function result.

```
void FailMemError();
```

FailMemError() calls the toolbox MemError() function, and if not <noErr>, throws an exception of the returned error code. Useful to call after a memory operation that does not return an error in itself.

```
void FailResError();
```

FailResError() calls the toolbox ResError() function, and if not <noErr>, throws an exception of the returned error code. Useful to call after a resource operation that does not return an error in itself.

```
void Fail();
```

Fail() always throws an exception of type <kUnknownExceptionErr> (999). You can use this as a last resort to throw an exception when you don't have a more meaningful error code. You should generally avoid doing this, since your users won't thank you for vague and unhelpful error messages.

```
void FailNILRes( void* aResPtr );
```

FailNILRes() is very similar to FailNIL(), but throws an exception of <resNotFound> (-192) instead. Use this when you want to test the Handle returned from a GetResource() call.

```
void FailSilent();
```

FailSilent() throws a silent exception- it works as normal but does not result in an error being

shown to the user. <userCanceledErr> (-128) is also treated as a silent error by default.

```
void FailParamErr( OSErr theErr );
```

FailParamErr() converts any error that is not <noErr> to an exception of type <paramErr> (-50). Very rarely needed.

```
void FailNILParam( void* aPtr );
```

FailNILParam() is like FailNIL(), except that the exception thrown is of type <paramErr> (-50). This is a useful function for sanity checking method parameters that must be valid pointers, handles or object references.

```
void FailNILErr( void* aPtr, OSErr err );
```

FailNILErr() allows you to specify what error code is thrown when a NIL pointer or Handle is encountered. This allows your code to be more specific about the reason for an error, which is good news for users.

Defining Error Codes:

Apple reserve all negative error codes and also codes 0 to 40 or so. MacZoop resrves all other positive error codes up to 999. If you want to define your own error codes for your own objects, you can use the values from 1000 to 32768.

To report errors to the user meaningfully, simply add an ‘Estr’ resource to your application with the same ID as the error code. This resource, which in ResEdit can be opened as a ‘STR ’ resource, is plain text that is added to the generic error stub “The command could not be completed because ”. Your text provides the explanation, and perhaps offers a solution to the user for solving the problem. See Error Handling for a more complete description.

Class Name: ZEventHandler	Based on: -
Type:	Framework Class
Description:	Helper object of ZApplication to fetch and dispatch events.

ZEventHandler is created by ZApplication and is used by it to do the actual work of fetching and dispatching events. You will very rarely need to use this class, and even more rarely subclass it.

Class Definition

```
class     ZEventHandler   : public ZComrade
{
protected:
     short          clicks;         // number of mouse-clicks in a series
     EventRecord    lastEvent;      // the event recovered from queue
     Boolean        inBackground;   // TRUE if app in background
     char           epPPCYokeDown;  // how many loops between WaitNextEvent() on PPC?


public:

     ZEventHandler();
     virtual ~ZEventHandler() {};

     virtual void    EstablishCurrentHandler();
     virtual void    GetAnEvent( EventRecord* theEvent,
                                 const short mask = everyEvent );
     virtual void    DispatchAnEvent( EventRecord* theEvent );
     virtual void    InstallApplescriptHandlers();
     virtual void    InstallAppleEventHandler( const AEEventClass pClass,
                                               const AEEventID pID );

     virtual void    HandleWindowUpdate( const WindowPtr theWindow );
     virtual void    HandleWindowActivate( const WindowPtr theWindow,
                                           const Boolean state );

protected:
     virtual void    CountClicks(const WindowPtr target, const long clickTicks,
                                 const Point globalMouse );
     virtual void    HandleMouseEvent( const EventRecord& theEvent );
     virtual void    HandleKeyEvent( const char theKey, const Boolean isAutoKey,
                                 short modifiers );
     virtual void    PassIdle();
     virtual void    HandleHLEvent( const EventRecord& theEvent);
     virtual void    HandleOSEvent( const EventRecord& theEvent);
     virtual void    DoBalloons( ZWindow* aWindow, const Point globMouse );

public:
     inline short    GetClicks(){ return clicks;};
     inline Boolean  InBackground(){ return inBackground;};
     inline void     GetLatestEvent( EventRecord* anEvent ) { *anEvent = lastEvent; };
     inline short    GetLatestModifiers() { return lastEvent.modifiers; };
};
```

Data Members

<**clicks**> isthe number of mouse clicks accumulated in a given double-click period
<**lastEvent**> is the event being processed
<**inBackground**> is TRUE if the application is currently suspended
<**epPPCYokeDown**> is used to fetch events more efficiently on PowerPC computers

Methods

```
virtual void    EstablishCurrentHandler();
```

Sets up the value of gCurHandler, which establishes the command chain

```
virtual void    GetAnEvent( EventRecord* theEvent,
                            const short mask = everyEvent );
```

Fetches the next event from the event queue and returns it in <theEvent>. The <mask> parameter can be used to filter out unwanted events of a certain type if required- this is the same mask parameter passed to WaitNextEvent. This method uses the current values set in ZApplication for the sleep time, and “yokes down” calls to WNE on Power PC machines so events are fetched less frequently than once per loop. This leads to better performance by avoiding excessive context switches.

```
virtual void    DispatchAnEvent( EventRecord* theEvent );
```

Dispatches the event passed in.

```
virtual void    InstallApplescriptHandlers();
```

Installs the standard Apple Events (the four required events).Apple events must beregistered before your application can receive them.

```
virtual void    InstallAppleEventHandler( const AEEventClass pClass,
                                          const AEEventID pID );
```

Installs a specific Apple Event handler for the event of the passed class and event ID. You can call this as needed to install your own events. Once so registered, such events will arrive in the HandleAppleEvent() method of ZCommander.

```
virtual void    HandleWindowUpdate( const WindowPtr theWindow );
```

Handles update events.

```
virtual void    HandleWindowActivate( const WindowPtr theWindow,
                                      const Boolean state );
```

Handles activation events.

```
virtual void    CountClicks(const WindowPtr target, const long clickTicks,
                            const Point globalMouse );
```

Counts the number of mouse clicks within the double-click period. This checks that the sequence of clicks are in the same window, and calls ZWindow’s ClickInSamePlace() method to

resolvethe click spatially. The result is set in the <clicks> data member, and returned by GetClicks().

```
virtual void    HandleMouseEvent( const EventRecord& theEvent );
```

Handles the mouseDown event.

```
virtual void    HandleKeyEvent( const char theKey, const Boolean isAutoKey,
                                short modifiers );
```

Handles the keyDown and autoKey events.

```
virtual void    PassIdle();
```

Calls the Idle() method of the command chain. See ZCommander.

```
virtual void    HandleHLEvent( const EventRecord& theEvent);
```

Handles high level events such as Apple Events. Events that are not apple events are passed back to ZApplication.

```
virtual void    HandleOSEvent( const EventRecord& theEvent);
```

Handles OS events such as suspend and resume.

```
virtual void    DoBalloons( ZWindow* aWindow, const Point globMouse );
```

Manages balloon help display.

```
short GetClicks();
```

Returns the number of mouse clicks in the double-click period.

```
Boolean    InBackground();
```

Returns whether the appliction is currently suspended or not.

```
void GetLatestEvent( EventRecord* anEvent );
```

Returns the event being processed.

```
short GetLatestModifiers();
```

Returns the modifier state of the event being processed.

Class Name: ZGrafState	Based on: -
Type:	User Class
Description:	Saves and restores state of Grafport

ZGrafState is a small object used to record and restore the graphics state of a grafPort. It is intended to be generally used as a stack object.

Class Definition

```
class ZGrafState
{
protected:
     CGrafPtr   port;
     GDHandle   dev;
     RgnHandle  clip;
     PenState   pen;
     RGBColor   fore;
     RGBColor   back;
     short      font;
     short      fSize;
     Style      fStyle;
     short      fMode;

public:
     ZGrafState();
     virtual ~ZGrafState();

     void Record();
     void Restore();
};
```

Data Members

The data members store various aspects of the graphics state of a port. They are protected and should not be accessed directly.

Methods

Constructor- calls Record();
Destructor- calls Restore();

```
     void Record();
```

Saves the current state of the grafPort into the data members.

```
     void Restore();
```

Sets the port back to the values stored in the data members.

Class Name: ZMenuBar	Based on: ZComrade
Type:	Framework Class
Description:	Handles the menubar, command mapping and menu management.

ZMenuBar manages all aspects of the menu bar, the menus it contains and the commands associated with them. It also provides a number of additional menu-related utilities.

Class Definition

```
class     ZMenuBar : public ZComrade
{
protected:
     short          mBarID;         // res ID of original MBAR resource
     short          mbCount;        // number of items in main bar at top level
     short          miSeed;         // index counter
     short          mHelpOffset;    // count of items in help menu
     short**        mBarH;          // Handle to menubar data during construction
     ZArray*        theMenuCmds;    // array of MenuCmd structs used to map commands
     ZArray*        theMenus;       // array of MenuInfRec structs
     char           menuCheckChar;  // character used for checking a menu item
     short          wmMenuID;       // ID of windows menu
     short          mBarHeight;     // saved menubar height when hidden
     MBarHiding     mbHiding;       // menubar hiding behaviour
     Boolean        rbPending;      // TRUE if menubar redraw called during dispatch
     Boolean        inDispatch;     // TRUE if currently in dispatch

public:

     ZMenuBar( const short barID ) : ZComrade() { classID = CLASS_ZMenuBar;
                                                  mBarID = barID; };
     virtual ~ZMenuBar();

     virtual void         InitMenuBar();

     virtual void         UpdateMenuBar();
     virtual void         ClickMenuBar( const Point mousePt );
     virtual void         DispatchCommand( const long mSelect );
     virtual void         DimMenus();
     virtual void         PrepareMenusForDisplay();

     virtual void         EnableCommand( const long cmd );
     virtual void         EnableCommand( const short menuID, const short itemID );
     virtual void         DisableCommand( const long cmd );
     virtual void         DisableCommand( const short menuID, const short itemID );

     virtual void         CheckCommand( const long cmd, const Boolean checkOnOff );
     virtual void         CheckCommand( const short menuID, const short itemID,
                                        const Boolean checkOnOff );
     virtual void         CheckCommand( const short menuID, Str255 itemString,
                                        const Boolean checkOnOff );
     virtual void         CheckCommandWithChar( const long cmd,
                                        const char checkChar );
     virtual void         CheckCommandWithChar( const short menuID, Str255 itemString,
                                        const char checkChar );
     virtual void         SetCommandText( const long cmd, Str255 aText );
     virtual void         SetCommandText( const short menuID, const short itemID,
                                        Str255 aText );
```

```
      virtual void         SetCommandText( const long cmd, const short strListID,
                                           const short strIndex );
      virtual void         SetCommandText( const short menuID, const short itemID,
                                 const short strListID, const short strIndex );
      virtual void         SetCommandTextStyle( const long cmd, Style aStyle );
      virtual void         SetTitleHilite( const short menuID, const Boolean state );
      virtual void         SetMenuDimming( const short menuID,
                                           const DimmingOptions dimOpts );

      virtual void         AppendMenuToBar( const short menuID );
      virtual void         RemoveMenuFromBar( const short menuID );
      virtual void         AppendStdItems( const short menuID,
                                           const short iType = appendDANames );
      virtual MenuHandle   FindMenuID( const short menuID );

      virtual short        AppendHelpItem( Str255 itemText );

// automatic "windows" menu handling:

      virtual void         NominateWindowsMenu( const short menuID );

      inline  void         SetCheckMarkChar( const char aChar );
      inline  char         GetCheckMarkChar();

// font, style and size menu utilities, can be called from any UpdateMenus():

      virtual void         UpdateStyleMenu( TEStyleRunInfo* runInfo );
      virtual void         UpdateStyleMenu( Style aStyle );
      virtual void         UpdateFontSizeMenu( TEStyleRunInfo* runInfo );

// showing and hiding the menubar:

      virtual void         ShowHideMenuBar( MBarHiding    mHiding, Point gMouseLoc );
      virtual void         ShowHideMenuBar( MBarHiding    mHiding );
      inline  MBarHiding   GetMenuBarVisState() { return mbHiding; };

      virtual void         SetZoomSourceToCommand( const long aCmd );
protected:

      virtual void         LoadMenus( const Boolean autoInstall = TRUE );
      virtual void         LoadMenu( const short menuID, Boolean isHMenu = FALSE,
                                     Boolean autoInstall = TRUE );
      virtual void         LoadCMNUMenu( const short menuID, Boolean isHMenu = FALSE,
                                     Boolean autoInstall = TRUE );
      virtual void         UnloadMenu( MenuHandle mH );
      virtual void         PredimMenu( MenuHandle theMenu );
      virtual void         ParseMenuItem( Str255 iText, long* aCmd );
      virtual void         FindMCmd( const long mSelect, MenuCmd* aCmd );
      virtual void         FindCommand( const long cmd, short* menuID, short* itemID );
      virtual long         TrackMenuBar( const Point mouse );
      virtual void         FindMenuInfo( const short menuID, MenuInfRec* mRec );
      virtual void         GetMenuTitleRect( short menuID, Rect* tRect );
};
```

Data Members

<**mbarID**> the resource ID of the ‘MBAR’ we are built from
<**mbCount**> the number of items listed in the ‘MBAR’ resource- this is not necessarily equal to the number of menus.
<**miSeed**>
<**mHelpOffset**> is the number of items originally in the Help menu. We need to know this to allow for this offset when dealing with commands attached to the help menu.
<**mbarH**> Handle to ‘MBAR’ resource during construction
<**theMenuCmds**> array of menu commands (MenuCmd structures)

<**theMenus**> array of menu info structures, one per menu
<**menuCheckChar**> character used to check a menu by default- usually the tick
<**wmMenuID**> the ID number of the “Windows” menu
<**mBarHeight**> the height of the original menubar when it’s hidden
<**mbHiding**> menu bar visible status
<**rbPending**> set TRUE if a menu title needs ot be redrawn
<**inDispatch**> set TRUE if a command is being dispatched to the command chain

Methods

```
virtual void        InitMenuBar();
```

Initialises the menu bar and all of the menus in it. It reads the ‘MBAR’ resource, adding all of the menus specified and all of the submenus that the main menus pull in. Commands are extracted and information is built into the two private data arrays, used later to provide command lookup. After creating the ZMenuBar object, this one call is all you need to completely set everything up.

```
virtual void        UpdateMenuBar();
```

Redraws the menubar when needed.

```
virtual void        ClickMenuBar( const Point mousePt );
```

Handles the mouse down event in the menu bar. Called by ZEventHandler as needed. This pulls down and tracks the menus, and dispatches the commands to the command chain.

```
virtual void        DispatchCommand( const long mSelect );
```

Looks up the command associated with the menu tracking result and sends it up the command chain.

```
virtual void        DimMenus();
```

Sets all menus to the dimmed state and removes all checkmarks (unless the menu has been specially marked not to do this). This is done just before dispatching the UpdateMenus() call to the command chain and pulling down the menu.

```
virtual void        PrepareMenusForDisplay();
```

Dims the menus using DimMenus() then calls the command chain with the UpdateMenus() message.

```
virtual void        EnableCommand( const long cmd );
virtual void        EnableCommand( const short menuID, const short itemID );
```

Enables the command or menu item passed (the menu item will appear available after this call).

```
virtual void        DisableCommand( const long cmd );
virtual void        DisableCommand( const short menuID, const short itemID );
```

Disables the command or menu item passed. The item will be greyed out after this call.

```
virtual void         CheckCommand( const long cmd, const Boolean checkOnOff );
virtual void         CheckCommand( const short menuID, const short itemID,
                                   const Boolean checkOnOff );
virtual void         CheckCommand( const short menuID, Str255 itemString,
                                   const Boolean checkOnOff );
```

Sets or clears a checkmark next to the command. This overloaded method can accept the command ID, the menu ID and item, or the menu ID and a string to match. Thi slast is useful to check menus with Fonts, etc, though mat be a little slow on older machines.

```
virtual void         CheckCommandWithChar( const long cmd,
                                   const char checkChar );
virtual void         CheckCommandWithChar( const short menuID, Str255 itemString,
                                   const char checkChar );
```

Sets the checkmark you pass next to the command or in the menu with the given ID and matched string.

```
virtual void         SetCommandText( const long cmd, Str255 aText );
virtual void         SetCommandText( const short menuID, const short itemID,
                                   Str255 aText );
virtual void         SetCommandText( const long cmd, const short strListID,
                                   const short strIndex );
virtual void         SetCommandText( const short menuID, const short itemID,
                              const short strListID, const short strIndex );
```

Sets the text of the menu item to the text specified. The item can be specified using the command number, or the menuID and item ID. The text string can be passed directly or looked up in a STR# resource with the passed ID and index.

```
virtual void         SetCommandTextStyle( const long cmd, Style aStyle );
```

Sets the style of the menu item associated with the command to the style passed, e.g. bold, italic, etc. Usually used for style menus.

```
virtual void         SetTitleHilite( const short menuID, const Boolean state );
```

Sets the title highlight of the given menu. This is usually used to cancel the highlight after a menu command has been processed.

```
virtual void         SetMenuDimming( const short menuID,
                                   const DimmingOptions dimOpts );
```

Sets the dimming options for a given menu. The dimming options specify how the menu is dimmed when the user clicks the bar. The options are:

```
enum
{
     neverDim        = 0,
     dimCommands     = 1,
     dimParentItems  = 2,
     dimOthers       = 4,
     dimAll          = 8,
     dimTitle        = 32
};
```

<**neverDim**> means that the entire menu state will remain as previously set by EnableCommand, etc. No automatic dimming is done at all.
<**dimCommands**> means that all items with associated command numbers will be dimmed.
<**dimParentItems**> means that menu items that are actually the “parent” items of a hierarchical submenu will be dimmed. The entire submenu associated will not be available unless the parent is explicitly re-enabled. Rarely used.
<**dimOthers**> means that all other items that are not commands or parent items will be dimmed.
<**dimAll**> means all items will be dimmed regardless of their status
<**dimTitle**> means that the title of the menu will be dimmed.

These flags can be combined to create combinations of behaviours for a particular menu.

```
virtual void         AppendMenuToBar( const short menuID );
```

Adds a menu to the menubar with the ID passed. The menu will be inserted before the Help menu, or after the last application menu of earlier systems. The menu can be a ‘MENUI’ or ‘CMNU’ resource, and take advantage of the full command handling ability of ZMenuBar. The menubar is redrawn by this method.

```
virtual void         RemoveMenuFromBar( const short menuID );
```

Removes a menu from the bar. This can be one added as above, or any other menu. Themenubar is redrawn accordingly. usually these two methods are used to implement dynamic menus that appear and disappear as a particluar window is activated and deactivated.

```
virtual void         AppendStdItems( const short menuID,
                                     const short iType = appendDANames );
```

This method allows menus to be filled with data according to some typical Apple schemes. If <iType> is **appendDANames**, the names of all the Apple Menu items are appaneded. This is done fo rthe Apple menu automatically. If <iType> is **appendFontNames**, thenames of the fonts are appended to the menu.

```
virtual MenuHandle   FindMenuID( const short menuID );
```

Returns the Mac menu handle for the menu with the given ID.

```
virtual short        AppendHelpItem( Str255 itemText );
```

Appends the text to the help menu, and returns the item number where the text was added.

```
virtual void         NominateWindowsMenu( const short menuID );
```

If your application wants a “Windows” menu, you can call this early on in your application to nominate the menu to use. ZMenuBar will then manage it entirely for you. Themenu can contain existing items if required, or be dedicated exclusively- it’s up to you. Themenu must have already been loaded from the ‘MBAR’ however.

```
inline  void         SetCheckMarkChar( const char aChar );
inline  char         GetCheckMarkChar();
```

Sets and returns the check mark character used for checking menus. Usually this will be the tick.

```
virtual void        UpdateStyleMenu( TEStyleRunInfo* runInfo );
virtual void        UpdateStyleMenu( Style aStyle );
```

If your application uses a “Style” menu, these methods are useful for maintaining it intelligently. When the user selects a given range of styled text, the Style menu should reflect the styles in the run. If there is more than one style in the run, the menu should indicate using multiple dash marks, only ticking those styles that exist throughout the selected run. Doing this properly requires quite a bit of information passed about the run, and that’s where <runInfo> comes in. To obtain this info, use TextStyleUtils function TEGetStyleRunInfo(). This takes a TextEdit handle and examines the internal state and generates the TEStyleRunInfo which you can pass here. This works provided you have set up your style menu using the standard commands for style.

The second version of the method is much simpler- it checks those styles that are flagged on in <aStyle>. This is usually only suitable for old-type TextEdit records that can only have a single global style. (Note- standard Text handling objects in MacZoop such as ZTextWindow do this for you automatically).

```
virtual void        UpdateFontSizeMenu( TEStyleRunInfo* runInfo );
```

This operates similarly to the above, but for the Font and Size commands. The runInfo record contains data about all three- font, size and style- in any styled run.

```
virtual void        ShowHideMenuBar( MBarHiding    mHiding, Point gMouseLoc );
virtual void        ShowHideMenuBar( MBarHiding    mHiding );
```

Show and Hide the menubar. You can hide and show it at will using MBAR_HIDE and MBAR_SHOW, or you can choose to implement automatic hiding (the menubar appears whenever the mouse strays near it). This is handled by the framework if you turn on the AUTO_MBAR_HIDING ProjectSetting switch.

```
inline  MBarHiding   GetMenuBarVisState() { return mbHiding; };
```

Returns whether the menubar is currenty visible or not.

```
virtual void        SetZoomSourceToCommand( const long aCmd );
```

Used as part of the support for the “zoom rects” visual effect- this sets up the global zoom sourcerect to the menu title for a command. usually you would not use this function directly- if you have the zoom effect enabled, it just works.

```
virtual void        LoadMenus( const Boolean autoInstall = TRUE );
```

Loads the menu resources as part of the initialisation.

```
virtual void        LoadMenu( const short menuID, Boolean isHMenu = FALSE,
                              Boolean autoInstall = TRUE );
```

Loads a menu and its submenus from a ‘MENU’ resource

```
virtual void        LoadCMNUMenu( const short menuID, Boolean isHMenu = FALSE,
                              Boolean autoInstall = TRUE );
```

Loads a menu and its submenus from a ‘CMNU’ resource. Note that the methods call each other

as needed as the submenus are read in, so it's fine to mix MENU and CMNU resources if you want.

```
virtual void         UnloadMenu( MenuHandle mH );
```

Gets rid of an installed menu.

```
virtual void         PredimMenu( MenuHandle theMenu );
```

Predims the menu according to the dimming flags for the menu.

```
virtual void         ParseMenuItem( Str255 iText, long* aCmd );
```

Extracts commands from the text of a menu item in a 'MENU' resource, returning the command number and the stripped text.

```
virtual void         FindMCmd( const long mSelect, MenuCmd* aCmd );
```

Looks up the MenuCmd data for the given menu tracking result passed.

```
virtual void         FindCommand( const long cmd, short* menuID, short* itemID );
```

This is the inverse- given a command, this looks up the menu ID and item ID associated with it.

```
virtual long         TrackMenuBar( const Point mouse );
```

Handles mouse tracking in the menubar.

```
virtual void         FindMenuInfo( const short menuID, MenuInfRec* mRec );
```

Looks up the menu info data for a given menu.

```
virtual void         GetMenuTitleRect( short menuID, Rect* tRect );
```

Returns the rectangle of the title of a given menu.

Comments

ZMenuBar is designed to be subclassable for particular requirements. While it naturally operates with the Mac OS menubar and menu handling, it is not required to do so. It is possible to use it to completely mimic the menubar behaviour in other situations, for example, in a window. That's why some of its methods break down more finely than you may expect at first. However, actually doing this is beyond the scope of this article- A ready made class may be offered at a later date.

Class Name: ZObject		Based on: -
Type:	Abstract Class; User	
Description:	Base class for most obejcts, allows objects to be streamable	

ZObject underpins most of MacZoop's classes. By having a common root, all objects are streamable using the same basic protocol.

Class Definition

```
class ZObject
{
protected:
     OSType     classID;        // class ID of object- subclasses must set this
     long       instanceID;     // instance ID. Reserved.

public:
     ZObject();
     virtual ~ZObject();

// info:
     void            GetClassName( Str255 aName );
     long            GetClassRef() { return classID; };
     long            GetInstanceID() { return instanceID; };
     void            SetClassID( OSType id ) { classID = id; };

// streaming:
     virtual void    ReadFromStream( ZStream* aStream ) {};
     virtual void    WriteToStream( ZStream* aStream ) {};

#if __OBJECT_DEBUG
     long            CountInstances( OSType ofClass = kAllClasses );
     virtual void    GetDebugInfoString( Str255 s );

     ZObject*   next;
     ZObject*   prev;
#endif
};
```

Data Members

<**classID**> is the unique four-character identifier for the class. For streamable ojects, this MUST be set.
<**instanceID**> a unique value for the individual instance of the object. At present, this is not used.
<**next**> is the next object in the debugging chain
<**prev**> is the previous object in the debugging chain

Methods

```
     void            GetClassName( Str255 aName );
```

Returns the class name of the object. This is the actual class name of the object, even if the object is subclassed.

```
long            GetClassRef();
```

Returns the class ID of the object.

```
long            GetInstanceID();
```

Returns the instance ID of the object. This is not used at present.

```
void            SetClassID( OSType id );
```

Sets the class ID of the object to <id>.

```
virtual void    ReadFromStream( ZStream* aStream );
virtual void    WriteToStream( ZStream* aStream );
```

General streaming methods. Subclasses may overridethese to stream their objects to the stream. The default methods do nothing. Subclasses should take care that the data is written and read in the same order.

```
long            CountInstances( OSType ofClass = kAllClasses );
```

Scans the debugging chain, counting the number of objects of a given class, or of all classes by default. The debugging chain is a doubly-linked list of all ZObject- derived objects that is maintained automatically if __OBJECT_DEBUG is set to 1. Inspectors and runtime debugging tools can examine this chain to find out about most of MacZoop's objects dynamically.

```
virtual void    GetDebugInfoString( Str255 s );
```

Returns readable information about the object for the benefit of inspectors and runtime debugging aids. Subclasses generally override this to provide useful info pertinent to the object.

Comments

For an object to be streamable, its class ID must be set and the class registered with gRegistry. If this is not done, it will not be possible to reanimate the object from the stream, because the actual type of the object will be not stored in the stream- the class ID is the vital piece of data that is put into the stream which allows the right type to be created when the stream is read in again.

It is the responsibility of objects that subclass MacZoop objects to define and set a proper unique Class ID. This is not done automatically. All MacZoop objects that can be streamed already do this. If you forget, what will happen is that the original MacZoop object will be created from the stream, not your subclass. Also, any extra data you put into the stream pertaining to your class will cause the stream to become misaligned and subsequently to fail.

See also: Streaming and Streaming macros

Class Name: ZObjectArray		Based on: ZArray
Type:	Template Container Class; User	
Description:	Generalised object storage class- can store arbitrary objects.	

ZObjectArray is an array that is optimised for storing object references. It is used in many places by MacZoop itself for storing various lists of objects. Use it wherever you need a list of objects of any type.

Class Definition

```
template <class T> class ZObjectArray : public ZArray
{
protected:
     T*** o;

public:
     ZObjectArray();

     virtual T*      GetObject( const long index );
     virtual void    SetArrayItem(void* item, const long index);
     virtual void    GetArrayItem(void* item, const long index);
     virtual void    ConcatenateArray( ZObjectArray<T>* anArray );
     virtual long    FindIndex(T* item);
     virtual void    Swap( const long itema, const long itemb );
     virtual void    DeleteObject(T* item);
     virtual void    DisposeAll();
     virtual Boolean Contains(T* item);
     virtual void    DoForEach( IteratorProcPtr aProc, const long ref );
     virtual void    MoveItem( const long curIndex, const long newIndex );
     virtual void    MoveToFront( const long index );
     virtual void    MoveToBack( const long index );
     virtual long    InsertSortedItem( void* item, SortCmpProcPtr compareProc,
                                       const long ref );

// streaming:

     virtual void    WriteToStream( ZStream* aStream );
     virtual void    ReadFromStream( ZStream* aStream );
};
```

Data Members

<**o**> is the correctly typed handle to the internal storage. Do not touch this!

Methods

```
     virtual T*      GetObject( const long index );
```

Returns the object stored at location <index>. Because this is a template class, the object will be correctly typed.

```
virtual void    SetArrayItem(void* item, const long index);
virtual void    GetArrayItem(void* item, const long index);
```

These methods override the similarly names ones in ZArray and perform the same function. However, they are optimised to operate more quickly than the ZArray methods, since the object size is fixed. Because ZObjectArray stores object references, <item> is actually a pointer to an object reference, or <T**>.

```
virtual void    ConcatenateArray( ZObjectArray<T>* anArray );
```

Joins two arrays. The array to join is passed in <anArray>. Th eobject references it contains are copied to the existing array, the joined array is unchanged.

```
virtual long    FindIndex(T* item);
```

Returns the index that the object reference <item> is found at. This is more efficient than ZArray, because only addresses are compared, not arbitrary data areas. However, the search is still inear.

```
virtual void    Swap( const long itema, const long itemb );
```

Swaps the position of objects at locations <itema> and <itemb>

```
virtual void    DeleteObject(T* item);
```

Deletes the object passed. The object is removed from the array but is not itself deleted. This calls FindIndex() to locate the object. Other items are moved down as neded to occupy the space vacated.

```
virtual void    DisposeAll();
```

Removes all objects from the array AND deletes them from memory. All objects are assumed to be heap objects, created using new. If this is not the case, do NOT call this method.

```
virtual Boolean Contains(T* item);
```

Returns TRUE if the array contains the object passed. This calls FindIndex().

```
virtual void    DoForEach( IteratorProcPtr aProc, const long ref );
```

Calls the Iterator proc for each object in the array. See ZArray for a full description of the iterator proc.

```
virtual void    MoveItem( const long curIndex, const long newIndex );
```

Moves the object at index <curIndex> to the position <newIndex>. Other items are moved as needed to accommodate the operation.

```
virtual void    MoveToFront( const long index );
virtual void    MoveToBack( const long index );
```

Moves the object at <index> to the front or back of the list respectively.

```
virtual long    InsertSortedItem( void* item, SortCmpProcPtr compareProc,
                                  const long ref );
```

Inserts an object into a sorted list using the comparison proc supplied to determinethe ordering. See ZArray for much more information about how this works. For his to work correctly, the list must be sorted.

```
virtual void    WriteToStream( ZStream* aStream );
virtual void    ReadFromStream( ZStream* aStream );
```

Streams the entire object array to the stream and back. Each object in the array is streamed, provided it is derived from ZObject.

Comments

ZObjectArray is used in two ways. Because it is defined as a template class, the usual way is to expand the templatefor each object type you wish to store. Here's how:

Suppose you want to define an object called MyObject, and use aZObjectArray to store a list of them. In your header:

```
// MyObject.h

#include  "ZObjectArray.h"

class MyObject;

typedef ZObjectArray<MyObject>  MyObjectList;

class MyObject : public ZObject
{
     // etc.....
};
```

You've now defined a container class called MyObjectList, which is a ZObjectArray that can contain only references to <MyObject>.Now you need to expand the template. In your implementation file:

```
// MyObject.cpp

#include  "MyObject.h"
#include  "ZObjectArray.cpp"

MyObject::MyObject()
     : ZObject()
{
     // etc....
}
```

Note that you #include the ZObjectArray.cpp file, NOT its header.

This approach is standard C++ and means that you have a correctly typed version of ZObjectArray which will accept and return MyObjects. This approach has a drawback however. Every time you declare a new array type like this, the code for it is duplicated. There is no way around this if you want correctly typed methods such as GetObject(). However, if you're concerned about the extra space occupied for each template expansion of ZObjectArray, and can accept that you'll need to typecast the result from GetObject(), etc, there is another way. In fact MacZoop itself creates its own lists this way to save space.

MacZoop predefines a container class as an expansion of ZObjectArray<ZObject>. It is called ZObjectList. Provided your own objects are derived from ZObject, the single expansion of ZObjectArray will suffice for all variants. The price you pay is the fact that methods such as GetObject() returns a ZObject*, so you need to typecast it to the true object type. i.e.

```
ZWindow* aWindow;

aWindow = (ZWindow*) myGenericList->GetObject( 2 );
```

ZObjectList is defined in the header of ZObjectArray, so you only need to include this header to make use of this general object class. You can mix both approaches if you wish.

Untyped parameters

With all ZObjectArrays, be careful about what you are actually getting when dealing with void* parameters. For example, comparison functions are declared to have void* parameters. What is actually passed is a POINTER to what the array is storing. The array is storing an object reference- another pointer. So to access the object itself in a comparison function, you have to dereference once and typecast, e.g.:

```
short MyCompareObjects( void* a, void* b, long ref )
{
      ZWindow*        wa;
      ZWindow*        wb;

      wa = (ZWindow*) *a;
      wb = (ZWindow*) *b;

      // now you can compare something in the objects....


}
```

It is a common error to forget to dereference the parameters, or go too far dereferencing them! Asimilar problem occurs with GetArrayItem. Here's how it should be used:

```
ZWindow* aWindow;

myWindowList->GetArrayItem( &aWindow, 2 );
```

Note the address operator- very important.

Class Name: ZTimer		Based on: ZComrade
Type:	Framework Class, utility object	
Description:	Provides regular timing to any ZCommander object	

ZTimer provides a simple way to provide regular callbacks to any ZCommander object. In addition, because ZTimer is a ZComrade, you can also listen to timers to get regular messages even if you are not a ZCommander object (you do have to be a ZComrade of course). See also: Timers

Class Definition

```
class ZTimer     : public   ZComrade
{
protected:
     long            id;
     unsigned long   interval;
     unsigned long   lastTicks;
     ZCommander*     wOwner;
     Boolean         oneShot;

public:

     ZTimer(    unsigned long tRate = kOneSecond,
                long anID = 0,
                ZCommander* owner = NULL,
                Boolean isOneShot = FALSE );

     inline long          GetID() { return id; };
     inline ZCommander*   GetOwner() { return wOwner; };
     inline long          GetElapsedTime() { return TickCount() - lastTicks; };
     inline void          SetRate( long tRate ) { interval = tRate; };

     virtual void         Do();
     virtual void         GetDebugInfoString( Str255 s );
};
```

Data Members

<**id**> is the timer's identifier.
<**interval**> is the time period of the timer in ticks (1/60th second)
<**lastTicks**> is used to calculate the time interval
<**wOwner**> is the commander that owns the timer
<**oneShot**> is TRUE if timer is a one-shot device, FALSE if periodic.

Methods

```
     ZTimer(    unsigned long tRate = kOneSecond,
                long anID = 0,
                ZCommander* owner = NULL,
```

```
                Boolean isOneShot = FALSE );
```

The constructor creates and sets up the timer in one go. <tRate> is the time period required, in ticks. The default is 1 second (60 ticks). <anID> is the timer ID number. If 0 (thedefault) an ID is assigned that will be unique. The id allows your commander to distinguish between multiple timers. <owner> is the ZCommander object to call back when the timer fires. This may be NULL. In which case, the timer will broadcast messages to any listeners instead. <isOneShot> sets the mode, either periodic (the default) or one shot. In one shot mode, the timer fires, sends its callback or message, then deletes itself.

```
long            GetID();
```

Returns the timer's ID number

```
ZCommander*     GetOwner();
```

Returns the timer's owner, if any.

```
long            GetElapsedTime();
```

Returns the elapsed time since the last time the timer fired, in ticks.

```
void            SetRate( long tRate );
```

Sets the timing interval. This method allows you to vary the rate dynamically if you so choose.

```
virtual void    Do();
```

Executes the timer's standard firing code.If you subclass ZTimer, you can override this to do whatever you want. The default method implements the ZCommander callback or ZComrade broadcast.

```
virtual void    GetDebugInfoString( Str255 s );
```

Returns readable info about the object for the benefit of debuggers, inspectors, etc.

Comments

You rarely will directly create an object of type ZTimer. This is because to work efficiently, the application implements a queue of timers which is iterated very quickly. Maintenance of this queueis best left to the experts, so usually you use the SetTimer() function to create and install the timer in one quick movement. Here are the timer utility functions:

```
ZTimer*    SetTimer( ZCommander* aCmdr = NULL,
                     long id = 0,
                     unsigned long interval = kOneSecond,
                     Boolean isOneShot = FALSE );
void       KillTimer( ZCommander* aCmdr, long id );
void       KillAllTimers( ZCommander* aCmdr );
```

SetTimer creates a ZTimer object with the parameters passed and installs it into the timer queue. Timing begins immediately- there is no priming required.
KillTimer removes the timer associated with <aCmdr> with the ID passed and deletes it.
KillAllTimers deletes all of the timers associated with a partcular commander- it is called automatically when a commander is deleted.

Creating your own timer class

This is very rarely required, but you can do it if needed. Subclass ZTimer and override the Do() method. Here you must check the time and then execute if the interval has elapsed. Do() will be called as often as possible. You must manually insert the timer into the timer queue. The queue is the global gTimerQ. You should append your timer using AppendItem()- gTimerQ is a standard ZObjectList.

Timer messaging

As mentioned, you do not have to associate a timer with a commander. By passing NULL as the owner, ZTimer will send messages when it fires instead of calling the owner's DoTimer() method. You can pick up these messages using the standard ReceiveMessage() method.

The message is defined:

```
enum
{
     kTimerMsgTimerTripped     = 'ttrp'
};
```

Note that ZTimer provides either/or. You won't get messages if the timer has an owner. To work around this, you can create two timers with the same interval, one with an owner and one without.

Timers can only fire if given time by the main event loop., If you hog the CPU and do not use a standard progress dialog, you may want to call TimerTimer() periodically to give timers time.

Class Name: ZWindow	Based on: ZCommander
Type:	User; Abstract Document Class
Description:	Implements basis for all windows, dialogs and documents.

ZWindow is the class common to all types of windows and dialogs in MacZoop. It provides both a drawable area on screen, as well as links to an associated file, drag and drop, etc. Please refer to the ZWindow chapter in the User Manual for full details.

Class Definition

```
class      ZWindow : public ZCommander
{
      friend class ZWindowManager;

protected:
      Rect            sizeRect;       // min and max sizes for window
      WindowPtr       macWindow;      // the mac window associated with this object
      short           windID;         // res ID of 'WIND' template
      Boolean         isNamed;        // TRUE if window has a name
      Boolean         stationeryFile; // TRUE if file opened was  stationery (template)
      FSSpec          macFile;        // the file that corresponds to this window
      OSType          macFType;       // file type last opened
      Boolean         printable;      // TRUE if this window can be printed
      Boolean         isPrinting;     // TRUE if draw operation is to printer
      Boolean         floating;       // TRUE if this is a floating window
      Boolean         disableAutoClose;// if TRUE, skip during Quit or Close All
      Rect            zoomSource;     // global rect where window was zoomed from
      RGBColor        winBackColour;  // original window background colour

private:
      Boolean         dirty;          // TRUE if window needs to be saved

public:
      ZWindow( ZCommander* aBoss, const short windowID );
      ZWindow();
      virtual ~ZWindow();

// initialisation (MUST be called after construction)
      virtual void          InitZWindow();

// drawing and clicking
      virtual void          Focus();
      virtual void          Draw();
      virtual void          DrawGrow();
      virtual void          Click( const Point mouse, const short modifiers);
      virtual void          AdjustCursor( const Point mouse, const short modifiers);
      virtual void          PostRefresh();
      virtual void          PostRefresh( Rect* aRect );
      virtual Boolean       ClickInSamePlace( const Point click1, const Point click2 );
      virtual void          SetDefaultColours();

// top-level call to do an update

      virtual void          PerformUpdate();
```

```
// window state manipulation
     virtual void         Hide();
     virtual void         Show();
     virtual void         Select();
     virtual void         Activate();
     virtual void         Deactivate();
     virtual Boolean      Close( const short phase );
     virtual Boolean      CloseSubsidiaryWindows( const short phase );
     virtual void         SendBehind( ZWindow* aWindow = NULL );

// command handling
     virtual void         HandleCommand( const long aCmd );
     virtual void         HandleCommand( const short menuID, const short itemID );
     virtual void         UpdateMenus();
     virtual void         SetTask( ZUndoTask* aTask );

// sizing and zooming
     virtual void         SetSizeRect( const Rect& szRect );
     virtual void         GetSizeRect( Rect* szRect );
     virtual void         Zoom( const short partCode );
     virtual void         PlaceAt( const short hGlobal, const short vGlobal );
     virtual void         Place();
     virtual void         PlaceRelative( ZWindow* relWindow, WindowPlacing aPlacing );

     virtual void         SetSize( const short width, const short height, const
                                   Boolean reDraw );
     virtual void         SetSize( const short width, const short height ) {
SetSize(width, height, TRUE); };
     virtual void         SetStdZoomRect( const Rect& aRect );
     virtual void         SetUserZoomRect( const Rect& aRect );
     virtual void         GetIdealWindowZoomSize( Rect* idealSize );
     virtual void         WindowResized() {};
     virtual Boolean      IsResizeable();

// file handling
     virtual Boolean      Save( const Boolean forceSaveAs = FALSE );
     virtual void         SaveFile();
     virtual void         Revert();
     virtual void         SetFile( const FSSpec& aFile );
     virtual void         OpenFile( const OSType aFileType,
                                    Boolean isStationery = FALSE );
     virtual void         PickFile( StandardFileReply* macReply );
     virtual void         SetDirty( Boolean dState );

#if _USE_NAVIGATION_SERVICES
     virtual void         PickFile( NavReplyRecord* navReply );
#endif

// print handling
     virtual void         CalcPages( const Rect& paperRect, short* pagesH,
                                                          short* pagesV );
     virtual void         PrintOnePage( const short pageNum, const Rect& paperRect );
     virtual void         PrintingStarting() { isPrinting = TRUE; };
     virtual void         PrintingFinishing() { isPrinting = FALSE; };

     inline    Boolean    IsPrintable() { return printable; };

// other info
     virtual void         SetTitle( Str255 aTitle );
     virtual void         GetName( Str255 name );
     virtual void         GetContentRect( Rect* contents );
     virtual void         GetBounds( Rect* aBounds ) { GetContentRect( aBounds ); };
     virtual Boolean      IsVisible();
     virtual Boolean      IsActive();
     virtual void         GetDebugInfoString( Str255 s );
```

```
// positioning and frame info:
     virtual short        GetTitleBarHeight();
     virtual void         GetStructureRegion( RgnHandle aRgn );
     virtual void         GetContentRegion( RgnHandle aRgn );
     virtual void         GetStructureFrameBorder( Rect* aRect );
     virtual void         GetGlobalPosition( short* hGlobal, short* vGlobal );

// saving and restoring window position as resource in file or prefs:

     virtual void         SavePosition( short id = 0 );
     virtual void         RestorePosition( short id = 0 );

// various inline getters & setters
     inline    WindowPtr  GetMacWindow(){ return macWindow; };
     inline    Boolean    Floats() { return floating; };
     inline    Boolean    NoAutoClose() { return disableAutoClose; };
     inline    void       GetFileSpec( FSSpec* aSpec ) { *aSpec = macFile; };
     inline    Boolean    IsDirty() { return dirty; };
     inline    OSType     GetFileType() { return macFType; };
     inline    void       GetBackColour( RGBColor* aColour );

// streaming:

     virtual void         WriteToStream( ZStream* aStream );
     virtual void         ReadFromStream( ZStream* aStream );

// drag and drop support:

     virtual Boolean      Drag( const Point startPt );
     virtual void         Drop( const OSType flavour, const Ptr data,
                                const long dataSize,
                                const DragReference theDrag = NULL ) {};
     virtual void         DragHilite( const Boolean state,
                                      const DragReference theDrag );
     virtual Boolean      AcceptsFlavour( const OSType aFlavour ) { return TRUE; };
     virtual RgnHandle    MakeDragRgn();
     virtual void         MakeDragData( const DragReference theDrag ) {};

// lowest level d+d handlers:

     virtual void         DragDispatch( const DragTrackingMessage theMessage,
                                        const DragReference theDrag);
     virtual void         DropHandler( const DragReference theDrag );

// help:

     virtual void         ShowBalloonHelp( Rect* localRect, Point tip,
                                           HMMessageRecord* hm );

protected:

// user methods
     virtual void         DrawContent();

// d+d installers

     virtual void         InstallDragHandlers();
     virtual void         RemoveDragHandlers();

// tracking handler methods

     virtual void         EnteredHandler( const DragReference theDrag) {};
     virtual void         EnteredWindow( const DragReference theDrag);
     virtual void         InWindow( const DragReference theDrag) {};
     virtual void         LeftWindow( const DragReference theDrag);
     virtual void         LeftHandler( const DragReference theDrag) {};
```

```
// constructing mac windows:

     virtual void         MakeMacWindow( const short windID );
     virtual void         MakeMacWindow( Rect* aRect, Str255 title,
                                         Boolean visible = FALSE,
                                         short varCode = 0,
                                         Boolean hasCloseBox = FALSE,
                                         void* userData = NULL );
     virtual void         InitSizeFromResource( ResType aType );
};
```

Data Members

<**sizeRect**> is the minimum and maximum limits that the window may shrink or grow to. The top and left fields set the minimums, and the bottom and right fields set the maximums.
<**macWindow**> is the Mac OS WindowPtr that this object is managing
<**windID**> is the resource ID of the original WIND template
<**isNamed**> is TRUE if the file has been named- i.e. it has a file on disk
<**stationeryFile**> is TRUE if the window was opened from a stationery file
<**macFile**> is the filespec of the associated file
<**macFType**> is the file type of the associated file
<**printable**> is TRUE if the window supports the Print command
<**isPrinting**> is TRUE if the contents are being rendered to a printer rather than the screen
<**floating**> is TRUE if the window is in the floating layer
<**disableAutoClose**> is TRUE if this window should be ignored when performing the CloseAll command.
<**zoomSource**> is a rectangle where the window zoomed from, if zooming effects are enabled
<**winBackColour**> is the original colour of the window background
<**dirty**> is TRUE if the data in the window should be saved before closing.

Methods

```
     ZWindow( ZCommander* aBoss, const short windowID );
     ZWindow();
     virtual ~ZWindow();
```

Constructors and destructor. Parameters to standard constructor are: <aBoss> the commander object that is the boss of the window. <windowID> is the resource ID of the WIND resource used to set up the basic window parameters, such as its style, size, etc.

```
     virtual void         InitZWindow();
```

Compulsory initialisation method. This method actually builds the window object structures and the associated Mac OS toolbox window that it manages. It must be called after creating the window object. This method is called separately because it is designed to be overridden for special window types (such as dialogs).

```
     virtual void         Focus();
```

Sets the window as the current port, and resets the clip region and port origin. A necessary early step in the handling of an update. Your own code can call this to establish this window as the current port. Note- MacZoop expects you to set the port as needed using this method before drawing. It does the same when it performs drawing operations. It is not necessary to go to long

lengths to preserve and maintain the current port- just set it and draw. As long as everyone follows this rule, there will never be any problem with drawing in the incorrect port, etc.

```
virtual void        Draw();
```

Handles the basic mechanism of handling an update. This method belongs to the framework- to draw the content of your own windows, overridethe DrawContent() method.

```
virtual void        DrawGrow();
```

Draws the grow box for windows that have one.

```
virtual void        Click( const Point mouse, const short modifiers);
```

Handles the basic mechanism of handling a mouse click in the interior part of the window. Th edefault method does nothing or else installs a standard ZMouseTracker object. Your own subclasses can overridethis to handle the mouse in the interior region.

```
virtual void        AdjustCursor( const Point mouse, const short modifiers);
```

Called whenever the mouse is over your window's interior region. You can determine what part of the window it's over and call SetCursorShape() accordingly.

```
virtual void        PostRefresh();
virtual void        PostRefresh( Rect* aRect );
```

Posts an update event for the entire window content area (upper) or for a sub-rectangle within it (lower). Subsequent event processing will cause your window to be updated. This is the usual mechanism for causing a window to refresh.

```
virtual Boolean     ClickInSamePlace( const Point click1, const Point click2 );
```

This is called by ZEventHandler to resolve a double-click in the window.Two sequential points are passed, you should determine whther these two points are in the "same place" logically speaking (e.g. the same button, same area of the window) and return TRUE if so. The default method simply returns TRUE, you usually override this to more finely resolve the double-click.

```
virtual void        SetDefaultColours();
```

Sets the window's port to black foreground colour and the window background colour. Rarely used.

```
virtual void        PerformUpdate();
```

High-level method to perform an update. This sets up the update region and calls Draw(). You can post an update using PostRefresh(), then call this immediately to process it if you do not wish to wait for the event to be fetched from the queue.

```
virtual void        Hide();
virtual void        Show();
```

Hides and Shows the window. Does not affect front-to-back ordering EXCEPT if the window is hidden while active- the next frontmost window will be selected in this case.

```
virtual void        Select();
```

Makes the window visible, and brings it to the front and activates it, making it the current window. This should be called as necessary after creating a window to make it available to the user.

```
virtual void        Activate();
virtual void        Deactivate();
```

Called when the user activates and deactivates the window. These methods are often overridden to change the appearance of items within the window as the activation state changes.

```
virtual Boolean     Close( const short phase );
```

Called when the user clicks in the Close box or chooses the Close command in the file menu. This is the recommended way to remove a window permanently. You should avoid deleting a window without calling this- if unavaoidable, call Hide prior to deleting the window object. Close() closes any child windows, then examines the save status. It asks the user if changes need to be saved, and handles the response. If the window is really closing, it is hidden then deleted.

```
virtual Boolean     CloseSubsidiaryWindows( const short phase );
```

Closes all of the child windows this window may be the boss of by calling their Close methods in turn. Becausethe user can cancel at any time, it may result in some windows being deleted but not all. Called by Close().

```
virtual void        SendBehind( ZWindow* aWindow = NULL );
```

Moves this window behind the passed on, generating activate events as needed. If the passed window is NULL, this window is moved behind all of the other windows. You can pass NULL to this method of the top window to implement a Cycle Windows command very easily.

```
virtual void        HandleCommand( const long aCmd );
virtual void        HandleCommand( const short menuID, const short itemID );
```

Overrides ZCommander’s method to implement the commands Save, Save As, Revert and Close.

```
virtual void        UpdateMenus();
```

Manages the display state of the commands it handles, namely Save, Save As, Revert and Close.

```
virtual void        SetTask( ZUndoTask* aTask );
```

Sets up an Undo task for this window. While it is ZApplication that manages the Undo menu command, the command is associted with a window. This is where an undo task is linked to the window. This also sets the dirty flag.

```
virtual void        SetSizeRect( const Rect& szRect );
virtual void        GetSizeRect( Rect* szRect );
```

Sets the SizeRect which defines the minimum and maximum sizes that the window can grow or shrink to. GetSizeRect() returns the current value. When setting the sizeRect, ifthe window does not currently conform it will be resized as required to make sure it does. This limit is honoured for growing and zooming. This value can be set up automatically when the window is created by

having a ‘WLIM’ resource with the same ID as the WIND resource. The WLIM is simply a rectangle that is used to initially set sizeRect.

```
virtual void         Zoom( const short partCode );
```

Called when the user clicks the zoom box of the window. This method deals with everything needed to implement intelligent zooming behaviour. It is rarely needed to override this method. You should, however, override the GetIdealWindowZoomSize() method to supply the Zoom function with the ideal size you’d like the window to be. Zoom sizes the window using the minimum movement while obeying the constraints of the monitor and the sizeRect to comply with your request. It also zooms the window so that it is fully on the monitor containing the largest area- for those of you with more than one monitor!

```
virtual void         PlaceAt( const short hGlobal, const short vGlobal );
virtual void         Place();
virtual void         PlaceRelative( ZWindow* relWindow, WindowPlacing aPlacing );
```

All three of these methods place the window on the monitor. PlaceAt putsthe window so that the top, left corner of its content area are positioned at <hGlobal> and <vGlobal>. The Place() method calculates the position based on windows it has placed in the past- in fact it defers to ZWindowManager to do this. PlaceRelative positions the window relative to an existing window. The<aPlacing> parameter specifies how it should be placed:

```
typedef enum
{
     kNoPosition,
     kCentreOnParent,
     kAlertPositionOnParent,
     kStaggerOnParent,
     kCentreOnScreen,
     kAlertPositionOnScreen,
     kStaggerOnScreen,
     kCentreOnParentScreen,
     kAlertOnParentScreen,
     kStaggerOnParentScreen
}
WindowPlacing;
```

Note these are MacZoop constants- not to be confused with similar ones in the Mac OS toolbox.

```
virtual void         SetSize( const short width, const short height, const
                              Boolean reDraw );
```

Sets the size of the window to the width and height requested, BUT constrained according to the sizeRect. If <reDraw> is TRUE, the interior is refreshed as needed.

```
virtual void         SetStdZoomRect( const Rect& aRect );
virtual void         SetUserZoomRect( const Rect& aRect );
```

Internal methods set up by the Zoom method. Do not call these directly. If you want to pre-zoom your window before it opens, call Zoom(inZoomOut).

```
virtual void         GetIdealWindowZoomSize( Rect* idealSize );
```

Your window class should override this to inform the zoom function the ideal size of the window. You need not be concerned with constraints such as monitors and sizeRect- Zoom will deal

with that. For example, if your window contains an array of icons like a Finder window, you could find the rectangle that encloses the icons, expand it a little bit, and return that. The benefit of this approach is that window zooming does what the user wants and does not simply make the window enormously large. Note that scroller windows (see ZScroller) return their bounds rect here which is usually the ideal size, so you only need to override this for windows based on ZWindow.

```
virtual void         WindowResized();
```

Called whenever the window size changes, for whatever reason. You can overridethis if you want to perform some operation here.

```
virtual Boolean      IsResizeable();
```

Returns TRUE if the window is resizeable. This is established from the style of the window originally set in the WIND template.

```
virtual Boolean      Save( const Boolean forceSaveAs = FALSE );
```

Handles the Save and SaveAs commands. This presents the file save dialog (using Navigation Services if available)and calls the SaveFile() method to actually write the file.

```
virtual void         SaveFile();
```

This method should be overridden to actually write your window's data model to the file in whatever form you wish. The default method performs somenecessary housekeeping and should be called after your own processing.

```
virtual void         Revert();
```

Handles the Revert command. After confirming with the user, this calls the OpenFile method to re-read the original file. Revert will only be vailable if a file was opened or saved and the window is dirty.

```
virtual void         SetFile( const FSSpec& aFile );
```

Sets the file this window is associated with. Usually called by ZApplication as needed.

```
virtual void         OpenFile( const OSType aFileType,
                               Boolean isStationery = FALSE );
```

This method should be overridden to read the associated file into your internal data model. SetFile() will have been called prior to this, so you should go ahead and open the file <macFile>. The default method does some necessary housekeeping, so you should call it after performing your own processing.

```
virtual void         PickFile( StandardFileReply* macReply );
virtual void         PickFile( NavReplyRecord* navReply );
```

Displays the standard Save File (or Navigation Services) dialog, and returns the file specification and other info about the file chosen. The window title is used as the initial file name choice.

```
virtual void         SetDirty( Boolean dState );
```

Marks the data in the window as requiring saving. You must always set the dirty flag using this method or SetTask(), which is why the dirty flag is private.

```
virtual void        CalcPages( const Rect& paperRect, short* pagesH,
                                                        short* pagesV );
```

Computes the pagination for printing the window. The paper rectangle is passed in <paperRect>, given the size of the window data content, you need to compute the number of pages vertically and horizontally and return them. The default method does this based on the bounds rectangle and paginates in row major order. See also ZPrinter.

```
virtual void        PrintOnePage( const short pageNum, const Rect& paperRect );
```

The default method manipulates the port origin and clip region and calls DrawContent() to render the printed view of the window. If your pagination scheme is different to standard, you will probably need to override this method too, since both assume the pagination scheme as described.

```
virtual void        PrintingStarting();
virtual void        PrintingFinishing();
```

The default methods simply set the <isPrinting> flag. You can override these to take additional action if needed.

```
inline    Boolean   IsPrintable() { return printable; };
```

Returns TRUE if the window can be printed.

```
virtual void        SetTitle( Str255 aTitle );
virtual void        GetName( Str255 name );
```

SetTitle sets the window’s title to the string passed. GetName returns the window’s title.

```
virtual void        GetContentRect( Rect* contents );
```

Returns the interior drawable area of the window (true for all window types, e.g. for ZScroler, this is the area less the scrollbars and margins).

```
virtual void        GetBounds( Rect* aBounds );
```

Returns the logical drawable area of the window. For ZWindow, this is the same as the content rect. For scrollers, it may be very different, and much larger.

```
virtual Boolean     IsVisible();
```

Returns TRUE if the window is visible.

```
virtual Boolean     IsActive();
```

Returns TRUE if the window is active.

```
virtual void        GetDebugInfoString( Str255 s );
```

Returns some readable information about the window that can be used by runtime debuggers and

inspectors.

```
virtual short        GetTitleBarHeight();
virtual void         GetStructureRegion( RgnHandle aRgn );
virtual void         GetContentRegion( RgnHandle aRgn );
virtual void         GetStructureFrameBorder( Rect* aRect );
```

These methods all return information about the window's frame. GetTitleBarHeight() returns the height of the titlebar, in pixels. GetStructureRgn() copies the structure region into the region passed (n.b. this works even when the window is hidden). GetContentRegion() does the same for the content region of the window. GetStructureFrameBorder() returns a rectangle structure whose fields contain the widths and heights of all four window frame edges- i.e. top contains the title bar height, left and right the left and right frame widths, and bottom the thickness of the window frame at the bottom. This information is used by the window manager when calculating window positions, etc.

```
virtual void         GetGlobalPosition( short* hGlobal, short* vGlobal );
```

Returns the global position of the top, left corner of the content area of the window. If the window is hidden, the result is unreliable.

```
virtual void         SavePosition( short id = 0 );
virtual void         RestorePosition( short id = 0 );
```

A quick method of saving and restoring the window's position. If the window has an associated file, the position is saved as a 'wpos' resource in that file, with the ID passed. If no associated file, any global preference file is used. If you need finer control over which file, etc is used, you can use ZWindowManager methods instead or override these to implement it in another way.

```
WindowPtr  GetMacWindow();
```

Returns the Mac OS windowPtr this object manages.

```
Boolean    Floats();
```

Returns TRUE if this is a floating window.

```
Boolean    NoAutoClose();
```

Returns TRUE if this window is ignored by the CloseAll command.

```
void       GetFileSpec( FSSpec* aSpec );
```

Returns the file spec of the file associated with the window. Note- if no file associated, the value of the <vRefNum> field of the file spec is set to kNoFile (-9999).

```
Boolean    IsDirty();
```

Returns TRUE if the window data needs to be saved.

```
OSType     GetFileType();
```

Returns the file type of the associated file.

```
void      GetBackColour( RGBColor* aColour );
```

Returns the original background colour of the window.

```
virtual void         WriteToStream( ZStream* aStream );
virtual void         ReadFromStream( ZStream* aStream );
```

Reads and Writes the complete window object to the stream.

```
virtual Boolean      Drag( const Point startPt );
```

Initiates a drag from the window. You need to override the MakeDragData and MakeDragRgn mtheods to add data to the drag.

```
virtual void         Drop( const OSType flavour, const Ptr data,
                           const long dataSize,
                           const DragReference theDrag = NULL ) {};
```

Handles a drop of dragged data on this window. This is called as many times as needed to handle all of the data items in the drag. <flavour> is the data type of the item, <data> is a pointer top its data, <dataSize> is its length, and <theDrag> is the original drag reference.

```
virtual void         DragHilite( const Boolean state,
                                 const DragReference theDrag );
```

The default method shows the drag highlight using the standard Drag manager calls. You can override it to do something fancy if you prefer.

```
virtual Boolean      AcceptsFlavour( const OSType aFlavour );
```

Overridethis method to inform the drag manager of the data types your window can accept. You will only be passed those types you pass back TRUE for. The default method accepts all data.

```
virtual RgnHandle    MakeDragRgn();
```

Create the drag outline for drags that originate in this window. The default method makes a drag region from the content rect of the window.

```
virtual void         MakeDragData( const DragReference theDrag );
```

Override this to add drag data to drags that originate in this window. use Drag Manager functions such as AddDataFlavor() to add the data (see Drag Manager reference guide).

```
virtual void         DragDispatch( const DragTrackingMessage theMessage,
                                   const DragReference theDrag);
virtual void         DropHandler( const DragReference theDrag );
```

These are low-level methods that handle the basic drag and drop mechanism. You will rarely need to override them, but you may if you need finer control over the drag and drop process. DragDisptach handles dragging into the window, DropHandler handles any subsequent dropping.

```
virtual void         ShowBalloonHelp( Rect* localRect, Point tip,
                                      HMMessageRecord* hm );
```

Handles the default display of balloon help messages for items in the window. If the

GetBalloonHelp method returns a valid help message and local rectangle, this method is called to display it. The default method displays help using HMShowBalloon().

```
virtual void         DrawContent();
```

This is where your window does its drawing. You should nearly always override this method. When called, the window port is established wit the correct clip region, origin, etc. Usually, you can just draw. Refer to the ZWindow chapter in the user manual for much more on this.

```
virtual void         InstallDragHandlers();
virtual void         RemoveDragHandlers();
```

Called by the initialisation code to set up the Drag manager. Do not use or override.

```
virtual void         EnteredHandler( const DragReference theDrag) {};
virtual void         EnteredWindow( const DragReference theDrag);
virtual void         InWindow( const DragReference theDrag) {};
virtual void         LeftWindow( const DragReference theDrag);
virtual void         LeftHandler( const DragReference theDrag) {};
```

These methods are all called by DragDispatch(). You will rarely need to use them. They may be overridden for special drag processing, but this would be unusual.

```
virtual void         MakeMacWindow( const short windID );
virtual void         MakeMacWindow( Rect* aRect, Str255 title,
                                    Boolean visible = FALSE,
                                    short varCode = 0,
                                    Boolean hasCloseBox = FALSE,
                                    void* userData = NULL );
```

Creates the Mac OS toolbox structures for the window. he first version creates the window from a WIND template, the second from a set of parameters. This version is called usually only when recreating a window from a stream. You should not have any cause to override these methods. ZDialog overrides them but the need to do so in an application is unlikely.

```
virtual void         InitSizeFromResource( ResType aType );
```

Initialises the sizeRect from an associated ‘WLIM’ resource. Note- this is called early in the initialisation of the window. It calls SetSizeRect() which MAY, if the window is currently not conforming to the limit, call SetSize(). Since the window is only partially initialised (any sub-class has not been set up yet) it is important that SetSize() and anything it may call is safe to call under these conditions. Mysterious crashes during start-up may be caused by this if you are not careful (other parts of MacZoop such as ZScroller are tolerant of calls to SetSize() during initialisation).

Class Name:	ZWindowManager
Type:	Framework Class
Description:	Handles layering and management of windows.

ZWindowManager is responsible for the overall management of windows in MacZoop, including maintaining the floating layer and other high-level features. It is created by ZApplication as a helper object, and the global <gWindowManager> contains its reference.

Class Definition

```
class      ZWindowManager  : public ZComrade
{
      friend class ZMenuBar;

protected:

      ZWindowList*    nonFloaters;         // list of non-floating windows
      ZWindowList*    floaters;            // list of floating windows
      ZWindowList*    wmWindows;           // list of windows in menu
      MenuHandle      wmMenu;              // handle of "Windows" menu if any
      short           wmItemOffset;        // item count of number of items in menu
      Point           globalPlaceLoc;      // placement position
      Rect            fStoredZoom;         // stored zoom for DeactivateForDialog
      Rect            fStoredZoomSource;   // ditto but source rect
      Boolean         wmActive;            // tracks active state of window manager
      Boolean         wmDDDeactivated;     // deactivated by non-MacZoop dialog

public:

      ZWindowManager();
      virtual    ~ZWindowManager();

      virtual void         AddWindow( ZWindow* aWindow );
      virtual void         RemoveWindow( ZWindow* aWindow );

      virtual void         HideWindow( ZWindow* aWindow );
      virtual void         ShowWindow( ZWindow* aWindow );

      virtual void         SelectWindow( ZWindow* aWindow );
      virtual void         DragWindowOutline( ZWindow* aWindow, Point startPt,
                                              const short modifiers );

      virtual void         Suspend();
      virtual void         Resume();
      virtual void         Deactivate();
      virtual void         DeactivateForDialog( short dlogID,
                                              Boolean isAlert = FALSE );
      virtual void         Activate();

      virtual ZWindow*     GetTopWindow();
      virtual ZWindow*     GetTopFloater();
      virtual ZWindow*     GetBottomFloater();

      virtual void         MoveWindowBehind( ZWindow* aWindow,
                                              ZWindow* behindWindow = NULL );
```

```
      virtual ZWindow*      LocateWindow( const Point globalMouse );
      virtual ZWindow*      GetNthWindow( const long n );
      virtual ZWindow*      GetNthFloater( const long n );
      virtual Boolean       IsDialog( ZWindow* aWindow );

      virtual Boolean       GetUniqueUntitledName( Str255 wName );
      virtual void          FloatIdle();

      virtual short         CountWindows();
      virtual short         CountFloaters();

      virtual void          InitiallyPlace( ZWindow* aWindow );
      virtual void          ZoomWindowClosed( ZWindow* aWindow );

// saving/restoring window positions

      virtual short         SaveWindowPosition( ZWindow* aWindow,
                                    ZResourceFile* aFile = NULL,
                                    short id = 0 );
      virtual void          RestoreWindowPosition( ZWindow* aWindow,
                                    ZResourceFile* aFile = NULL,
                                    const short id = 0 );

      virtual void          StackWindows(   Boolean resize = TRUE,
                                    Boolean mainList = TRUE,
                                    Boolean floaterList = FALSE );
      virtual void          TileWindows( Boolean verticalPreferred = TRUE );

private:
      void                  BringBehind( ZWindow* aWindow, ZWindow* behindWindow );
      void                  PostActivation( ZWindow* aWindow, Boolean state );
      void                  CalcWindowRgns( ZWindow* aWindow, RgnHandle aRgn );
      void                  ShowHideFloater( ZWindow* aFloater, Boolean hide );
      Boolean               WindowOnDesktop( Rect* wFrame );

protected: // these methods accessible to gMenuBar, but not user's code.

      virtual void          SetWindowsMenu( MenuHandle aMenu );
      virtual void          SelectWindowFromMenu( const short itemID );
      virtual void          BuildWindowsMenu();

      virtual Boolean       CommandClickInFrontDragBar( ZWindow* target,
                                                const Point startPt );
};
```

Data Members

<**nonFloaters**> list of non-floating windows in front-to-back order
<**floaters**> list of floating windows in front-to-back order
<**wmWindows**> list of windows in “Windows” menu, in menu order
<**wmMenu**> handle to the “Windows” menu
<**wmItemOffset**> number of items in “Windows” menu originally
<**globalPlaceLoc**> position of next window to be placed
<**fStoredZoom**>, <**fStoredZoomSource**> store the zoom effect rectangles when dialogs and alerts are displayed that are not MacZoop objects
<**wmActive**> TRUE if window manager active
<**wmDDDeactivated**> TRUE if deactivated by dialog that is not MacZoop object

Methods

Note: many methods in ZWindowManager are called internally. There are relatively few that

application code will normally make use of. Bear this in mind when reading these descriptions. In most cases, ZWindowManager methods are indirectly called by sending a message directly to the window concerned. You must take care not to go around this design because you will lose the advantage of being able to override methods at the window class level.

```
ZWindowManager();
virtual    ~ZWindowManager();
```

Constructor and destructor. Initialise the data members and create the lists.

```
virtual void         AddWindow( ZWindow* aWindow );
virtual void         RemoveWindow( ZWindow* aWindow );
```

These methods are called by ZWindow when the window is created and destroyed. The window is added and removed from the appropriate list with ZWindowManager. Normally, you will never need to call these methods, EXCEPT if you have a completely new kind of ZWindow object that overrides the standard code in InitZWindow(). After initialisation, the window MUST be added to the window manager with AddWindow(), and removed when deleted.

```
virtual void         HideWindow( ZWindow* aWindow );
virtual void         ShowWindow( ZWindow* aWindow );
```

Hides and Shows the window passed. Normally, you must call the window itself to do this.

```
virtual void         SelectWindow( ZWindow* aWindow );
```

Activates and makes visible the window passed. You should call the window's Select() method instead.

```
virtual void         DragWindowOutline( ZWindow* aWindow, Point startPt,
                                        const short modifiers );
```

Drags an outline of the window and move it accordingly. This also handles other click processing such as command-click in the title bar. Do not call this- it is called as required by the framework.

```
virtual void         Suspend();
virtual void         Resume();
```

The applocation is being suspended or resumed. ZWindowManager responds by hiding and showing the windows in the floating layer and generating activate events as needed.

```
virtual void         Deactivate();
```

All windows are deactivated. This is called when a modal dialog is displayed. You should not usually call this yourself.

```
virtual void         DeactivateForDialog( short dlogID,
                                          Boolean isAlert = FALSE );
```

This method may be called if you wish to display a dialog or alert that is not a MacZoop dialog (ZDialog) object. This would be rare, but some alerts, such as the About box, are like this. This method makes sure that the windows are properly deactivated, and also handles the zoom effect for the displayed window, if enabled.

```
virtual void        Activate();
```

Reactivates the windows after being deactivated by either of the Deactivate methods. You only need to call this if you previously manually deactivated the windows for a non-MacZoop dialog or alert.

```
virtual ZWindow*    GetTopWindow();
virtual ZWindow*    GetTopFloater();
virtual ZWindow*    GetBottomFloater();
```

These methods return the current top window, top floater and bottom floaters. For the top window, it is usually more convenient to call ZApplication's GetFrontWindow() method.

```
virtual void        MoveWindowBehind( ZWindow* aWindow,
                                      ZWindow* behindWindow = NULL );
```

Moves <aWindow> behind <behindWindow>, or to the back if <behindWindow> is NULL. Normally you would call the window's SendBehind() method directly.

```
virtual ZWindow*    LocateWindow( const Point globalMouse );
```

Returns the window that contains the global point passed. This is called to resolve the target of mouse clicks and cursor adjustment calls.

```
virtual ZWindow*    GetNthWindow( const long n );
virtual ZWindow*    GetNthFloater( const long n );
```

Returns the nth window in the main or floating layers respectively. Windows are numbered from 1 meaning the frontmost to the value returned by CountWindows() or CountFloaters().

```
virtual Boolean     IsDialog( ZWindow* aWindow );
```

Returns TRUE if the window is a MODAL dialog. Modeless dialogs and other windows return FALSE.

```
virtual Boolean     GetUniqueUntitledName( Str255 wName );
```

This method checks <wName> against the titles of the windows in the main layer. If the name is found to match an existing name, 1, 2, etc is appended to the name until it is unique. If the name is modified, it returns TRUE, if the name remains unchanged, it returns FALSE.Case and diacritical marks are ignored when comparing names. This is called automatically when a window is created to ensure the title is unique, thus conforming to the user interface guidelines. n.b. the is no reason why the name passed needs to be "untitled", though it often is- this works even if your window name has been localised or otherwise changed.

```
virtual void        FloatIdle();
```

This method ensures that floating windows get Idle time. Note that because floaters do not form part of the command chain, even though they are commanders, a special call is needed to give them idle time. Called by the framework as needed.

```
virtual short       CountWindows();
virtual short       CountFloaters();
```

Returns the number of windows in the main and floating layers respectively. The value returned is the maximum index number of a window in that layer.

```
virtual void         InitiallyPlace( ZWindow* aWindow );
```

Called by ZWindow's Place() method. This places the window on the main monitor cascade-wise from the top, left corner. The frame and titlebar thickness are taken into account when calculating the position. The cascade position is reset if all windows are closed. This method does nothing for floating windows- to place a floating window, use its PlaceAt() or PlaceRelative() method.

```
virtual void         ZoomWindowClosed( ZWindow* aWindow );
```

This method performs the zoom rect animation when a widnow is closed, if this feature is anabled. It will be called automatically by ZWindow's Close() method. To allow speedier quitting, this does nothing if the application phase is kQuitting.

```
virtual short        SaveWindowPosition( ZWindow* aWindow,
                                         ZResourceFile* aFile = NULL,
                                         short id = 0 );
virtual void         RestoreWindowPosition( ZWindow* aWindow,
                                         ZResourceFile* aFile = NULL,
                                         const short id = 0 );
```

These methods Save and Restore a window's position to the given resource file. Usually, you can use ZWindow's SavePosition() and RestorePosition() methods, but calling these methods give you a greater degree of control, since it allows you to specify the file explicitly. If <aFile> is NULL, the global gPrefsFile, if it exists, is used. The position info is saved as a 'Wpos' resource with the id passed. (If the window is a dialog, a 'Dpos' resource is created.). The RestoreWindowPosition method checks that the returned position is actually valid on the user's system. This prevents windows from being positioned off screen if the user changes the monitor configuration. See also: ZResourceFile.

```
virtual void         StackWindows(   Boolean resize = TRUE,
                                     Boolean mainList = TRUE,
                                     Boolean floaterList = FALSE );
```

Stacks the windows in cascade fashion on the main screen. This is called for you if your application has the standard 'Stack Windows" command. If <reSize> is TRUE, the window will be resized to an appropriate size, if FALSE, its existing size is used. if <mainList> is TRUE, the main windows are stacked. If <floaterList> is TRUE, the floaters are stacked in their own separate stack.

```
virtual void         TileWindows( Boolean verticalPreferred = TRUE );
```

Tiles the windows on the main screen. Only the main windows can be tiled, floaters are ignored. This method tries to arrange the windows as optimally as possible, leaving the minimum amount of unoccupied space on the monitor. If the number of windows is fewer than four, <verticalPreferred> sets whether the windows are tiled vertically (TRUE) or horizontally.

All other methods are protected or private. The "Windows" menu handling is accessed via ZMenuBar, though ZWindowManager is responsible for doing some of the work.

Class Name: ZDialog	Based on: ZWindow
Type:	User Class
Description:	Basis for any dialog box. May not need to be subclassed for straightforward dialogs.

ZDialog implements all types of dialog box in MacZoop. It retains all of the features and properties of ZWindow, but provides standard user-interface elements (each element is an object of type ZDialogItem). ZDialog is designed so that for most applications, it dos not require subclassing, though you are free to if you wish.

Class Definition

```
class     ZDialog : public ZWindow
{
friend class ZDialogItem;

protected:
     ZDialogItemList*     itsItems;       // list of item objects in ID order
     ZDialogItem*         focusItem;      // current handler
     Boolean              isModal;        // TRUE if dialog is modal
     Boolean              isInline;       // TRUE if dialog inline
     short                ditlID;         // ID of base DITL resource
     short                signalDismiss;  // item to close dialog
     short                exitItem;       // item that closed dialog
     short                baseItems;      // count of "native" items in the dialog
     short                focusCount;     // cache of focus count
     short                defaultItem;    // item the return/enter maps to
     short                escapeItem;     // item that escape/cmd-. maps to
     WindowPlacing        autoPos;        // automatic positioning requested
     Rect                 oldBounds;      // bounds before resize op

public:

     ZDialog( ZCommander* aBoss, const short dialogID );
     ZDialog();
     virtual ~ZDialog();

     // window handling stuff

     virtual void              InitZWindow();
     virtual void              Draw();
     virtual Boolean           Close(const short phase);
     virtual void              AdjustCursor(const Point mouse, const short modifiers);
     virtual void              Click( const Point mouse, const short modifiers );
     virtual void              Activate();
     virtual void              Deactivate();
     virtual void              Type( const char theKey, const short modifiers );
     virtual Boolean           GetBalloonHelp( const Point mouse,
                                               Rect* br,
                                              Point* tip,
                                              HMMessageRecord* hm );

     virtual void              SetSize( const short width, const short height,
                                        const Boolean reDraw );
     virtual void              Zoom( const short partCode );
```

```
// command stuff

virtual void              UpdateMenus();

// std edit commands

virtual void              DoCut();
virtual void              DoCopy();
virtual void              DoPaste();
virtual void              DoClear();

// dialog handling stuff

virtual void              SetUp();
virtual void              ClickItem( const short theItem );
virtual Boolean           CloseDialog();
virtual void              DrawOneItem( const short item );
virtual void              DrawUserItem( const short item, Rect* bounds );
virtual void              SetDialogBaseFont(   short fontID = 0,
                                               short fontSize = 12,
                                               short fontStyle = 0 );
virtual void              Place();

// convenience functions

virtual void              SetValue( const short item, const long value );
virtual void              SetValue( const short item, const int value );
virtual void              SetValue( const short item, const short value );
virtual void              SetValue( const short item, const Str255 value );
virtual void              SetValue( const short item, const double value );
virtual void              SetValue( const short item, const float value );

virtual long              GetValue( const short item );
virtual void              GetValueAsText( const short item, Str255 aStr );
virtual float             GetValueAsFloat( const short item );

virtual short             GetSelectedItemInGroup( const short groupID );

// info about dialog items:

virtual void              GetItemBounds( const short item, Rect* bounds );
virtual short             GetItemType( const short item );
virtual short             FindItem( const Point localMouse );
virtual void              FakeClick( const short item );
virtual Boolean           ValidateFields( Boolean showAlert = TRUE );

// manipulating items' appearance and behaviour:

virtual void              HideItem( const short item );
virtual void              ShowItem( const short item );
virtual void              EnableItem( const short item );
virtual void              DisableItem( const short item );
virtual void              SetItemTitle( const short item, Str255 title );

// changing the user's focus:

virtual void              SelectItem( const short item );
virtual void              SelectNextFocus();
virtual void              SelectPreviousFocus();
virtual short             CountFocusableItems();

virtual ZDialogItem*      GetItemObject( const short item );
virtual ZDialogItem*      GetDefaultItemObject();
virtual ZCommander*       GetHandler();
```

```
      // multi-part dialogs:

      virtual void              AppendItemsToDialog( const short ditlID,
                                                     DITLMethod apMethod = overlayDITL );
      virtual void              RemoveAppendedItems();
      inline  short             GetBaseItemCount();

      // streaming:

      virtual void              WriteToStream( ZStream* aStream );
      virtual void              ReadFromStream( ZStream* aStream );

      // d+d:

      virtual Boolean           AcceptsFlavour( const OSType aFlavour );
      inline  Boolean           IsModal();

      // convenience methodS for implementing "inline" modal dialogs- use with care!

      virtual Boolean           RunModal();
      virtual void              DismissModal( const short itemDismiss );
      virtual void              WindowResized();
      virtual void              SetItemSizing( const short item,
                                               unsigned char sizing );
      inline    short           GetDITLID();
      inline    short           GetDLOGID();

protected:

      virtual void              MakeMacWindow( const short dialogID );
      virtual void              MakeMacWindow( Rect* aRect,
                                               Str255 title,
                                               Boolean visible = FALSE,
                                               short varCode = 0,
                                               Boolean hasCloseBox = FALSE,
                                               void* userData = NULL );
      virtual void              ParseRButtonTitle(   Str255 buttonTitle,
                                                     short* groupID,
                                                     Boolean* isDefault );
      virtual void              HandleRButtonGroupClick( const short item );
      virtual void              ClearDITLPlaceHolders( Handle ditl );
      virtual Boolean           PasteDataIsLegal( const short targetItem );
      virtual void              ParseEditFieldInfo( Str255 efText,
                                                    unsigned short* efFlags,
                                                    long* min, long* max );
      virtual ZDialogItem*      MakeItemObject( const short item,
                                                const long magicType = 0 );
      virtual void              BuildDialogObjects( short fromItemNo = 1,
                                                    const short ictbID = 0 );
      virtual void              ParseStatText( Str255 sText,
                                               OSType* typeParam,
                                               short* paramCount,
                                               long params[] );
      virtual void              ClipOutItemsBelow( const short item );
      virtual void              UserInitialise( ZDialogItem* theItem ) {};
      virtual   long            DMToMagicType( short dmType );
      virtual void              InitItemFromICTB(    const short item,
                                                     ictbHandle ictb,
                                                     const short ictbIndex );
      virtual void              InitItemSizingFromRes( const short resID );
};
```

Data Members

<**itsItems**> list of ZDialogItems- the items in the dialog
<**focusItem**> the item that currently has the user's focus
<**isModal**> TRUE if the dialog is modal (blocks command chain)
<**isInline**> TRUE if the dialog is being run inline (synchronously)
<**ditlID**> is the resource ID of the original DITL resource used to build the dialog items
<**signalDismiss**> when set to an item number other than 0, will cause an inline dialog to close.
<**exitItem**> item number of the item that closed the dialog
<**baseItems**> count of items in the dialog before any were appended
<**focusCount**> count of focusable items in the dialog
<**defaultItem**> the item number of the item hit when return and enter are typed
<**escapeItem**> the item number of the item hit when the escape key is typed
<**autoPos**> used to position dialog initially
<**oldBounds**> stores the size of the window before it was changed- used to recalculate item positions.

Methods

```
ZDialog( ZCommander* aBoss, const short dialogID );
ZDialog();
virtual ~ZDialog();
```

Constructors create the basic object and reset the data members. Destructor disposes any memory allocated by the object.

```
virtual void              InitZWindow();
```

Main initialisation call- MUST be called after creating the dialog. This does everything needed to set up the dialog. It makes the underlying Mac dialog structures, reads in the items from the DITL resource, makes all of the various ZDialogItem objects and performs all necessary steps to initialise the dialog. After this call, the dialog is ready to use.

```
virtual void              Draw();
```

Overrides the ZWindow Draw method to draw each dialog item in response to an update event.

```
virtual Boolean           Close(const short phase);
```

Overrides the ZWindow method to handle the closure of the dialog. Calls the inherited method for non-inline dialogs so dialogs are still able to handle the "Save Changes?" alert, etc as any window.

```
virtual void              AdjustCursor(const Point mouse, const short modifiers);
```

Sets the cursor shape according to the item the cursor is over. This passes on the call to the dialog item itself to process, unless no item is under the mouse, in which case the cursor is reset.

```
virtual void              Click( const Point mouse, const short modifiers );
```

Overrides the ZWindow method to detect and handle mouse clicks in the dialog items. This click is passed to the relevant item for processing.

```
virtual void            Activate();
virtual void            Deactivate();
```

Overrides the ZWindow methods to respond to activate and deactivate events. The default behaviour is to disable all items on a deactivate, and re-enable them on an activate. The previous state of the item is recorded so that it does the right thing!

```
virtual void            Type( const char theKey, const short modifiers );
```

Responds to the keyboard at the dialog level (n.b. the focussed item begins the command chain and will handle its own keyboard input). The dialog responds to the tab and shift-tab keys by cycling through the focusable items. It also handles the return/enter and escape keys by mapping them to hits on the relevant items.

```
virtual Boolean         GetBalloonHelp( const Point mouse,
                                         Rect* br,
                                        Point* tip,
                                        HMMessageRecord* hm );
```

Returns balloon help information about dialog items by requesting the item under the mouse to return its balloon help information.

```
virtual void            SetSize( const short width, const short height,
                                 const Boolean reDraw );
```

Overrides the ZWindow method to allow dialog items to be repositioned automatically when the window size is changed. The inherited method is called so works as normal for the dialog window itself.

```
virtual void            Zoom( const short partCode );
```

As above for the zoom command. n.b. these methods just record the current window size in <oldBounds>, then call the inherited methods. The actual item repositioning is done by WindowResized().

```
virtual void            UpdateMenus();
```

Overrides ZCommander's UpdateMenus(). This defers to the standard ZWindow processing as long as the dialog is not modal. If it is modal, this method does nothing. This ensures that the menubar is inacessible when a modal dialog is up (except for Help items, etc). Because the focusable item starts the command chain, other commands such as Cut, Copy, Paste etc may be available if the individual item is able to accept them.

```
virtual void            DoCut();
virtual void            DoCopy();
virtual void            DoPaste();
virtual void            DoClear();
```

Overrides the ZCommander methods to handle the standard edit commands. These use the Dialog Manager functions DialogCut, etc. The dialog handles these commands on behalf of edit field items. Other items may handle the commands in their own way.

```
virtual void            SetUp();
```

This method is the last call in the initialisation stage. You can override it to do further set up for

your own dialog class.

```
virtual void              ClickItem( const short theItem );
```

This method responds to hits in the dialog item. It is called whenever the item is hit or its focus changes. The default method responds by implementing the standard dialog behaviours for the item type, including flipping the state of checkboxes, handling selection of buttons in a radio group, and so forth. You may call this to cause the action that a mouse click in the item would normally have (use FakeClick for a button, as it provides additional visual feedback).

```
virtual Boolean           CloseDialog();
```

Called when the dialog is going to close. This method can prevent closure by returning FALSE. If it returns TRUE, the dialog will close normally. The default method calls ValidateFields, and prevents closure if a field is invalid.

```
virtual void              DrawOneItem( const short item );
```

Draws a single dialog item by calling its Draw() method.

```
virtual void              DrawUserItem( const short item, Rect* bounds );
```

Provides a standard behaviour for user items. Note that in ZDialog, user items are really a thing of the past, since you can implement custom dialog items much more easily by subclassing ZDialogItem. The traditional user item is reserved for drawing group frames and boxes. This method implements this default behaviour. If you wish to use the old user-item model for handling items, you can override this method, but the object approach is strongly recommended and very superior!

```
virtual void              SetDialogBaseFont(   short fontID = 0,
                                               short fontSize = 12,
                                               short fontStyle = 0 );
```

This method allows you to define a dialog-wide font, style and size. Individual dialog items also have their own font settings which overridethis. If you do not set up an item's font individually (either manually or using an 'ictb' resource), the item will draw using the dialog's font. This method, if used, should be called after InitZWindow, but before the dialog is made visible.

```
virtual void              Place();
```

Overrides the ZWindow method to place the dialog according to the placement specified in the original 'DLOG' template.

```
virtual void              SetValue( const short item, const long value );
virtual void              SetValue( const short item, const int value );
virtual void              SetValue( const short item, const short value );
virtual void              SetValue( const short item, const Str255 value );
virtual void              SetValue( const short item, const double value );
virtual void              SetValue( const short item, const float value );
```

SetValue sets the <value> of the <item>. In general, it will do the right thing- pass any data type- integer, string, float, etc and get the expected result according to the item's type. This is called to set up the initial state of items, or at any time when you want to set an item programmatically. Note this method does not set control items' titles. Use SetItemTitle() for that.

```
virtual long            GetValue( const short item );
virtual void            GetValueAsText( const short item, Str255 aStr );
virtual float           GetValueAsFloat( const short item );
```

These methods are the inverse of SetValue. They return the item's value as the data type requested. Any conversions are done as needed (such as from string to float, etc). These methods are the usual way to extract data from a dialog user interface into a data structure or whatever.

```
virtual short           GetSelectedItemInGroup( const short groupID );
```

Returns the item number of the radio button that is ON in any particular group. This is convenient for determining what is selected in a group rather than looking at a number of items to see which one is ON.

```
virtual void            GetItemBounds( const short item, Rect* bounds );
```

Returns the bounds rectangle of the item in the window. The bounds rectangle completely encloses the interior of the item, but not any decorative border it may have.

```
virtual short           GetItemType( const short item );
```

Returns the Dialog Manager type of the item. In addition, custom items set the top bit. You should use this with caution, since the original Dialog Manager type of an item may mislead you when it comes to correctly dealing wit the item. Better to use the extended type, obtained using the item's GetXType() method.

```
virtual short           FindItem( const Point localMouse );
```

Returns the item number of the item under the mouse point passed, or 0 if none.

```
virtual void            FakeClick( const short item );
```

Simulates a mouse click on a button by highlighting the button for a short interval (8 ticks) and calling ClickItem().

```
virtual Boolean         ValidateFields( Boolean showAlert = TRUE );
```

Verifies the contents of editable text fields. This uses special flags set up for editable text fields (see ZDialogItem) to check that a numeric entry is within the valid range. If not, an alert is shown explaining the problem (unless <showAlert> is FALSE). The offending field is selected and the function returns FALSE. If all fields are OK, it returns TRUE.

```
virtual void            HideItem( const short item );
virtual void            ShowItem( const short item );
```

Hides and Shows a dialog item.

```
virtual void            EnableItem( const short item );
virtual void            DisableItem( const short item );
```

Enables and Disables a dialog item. A number of special behaviours are available. If <item> is 0, all items in the dialog are enabled or disabled (with state history recorded so that previously disabled items are not falsely re-enabled). This is done when the dialog is activated and deacti-

vated. If the value of <item> is negative, it is interpreted as a group ID and all items in that group will be enabled or disabled, again with the state history recorded. Because of the state history, you should take care to match enable and disable calls- if you disable twice, you'll need to call enable twice to compensate.

```
virtual void              SetItemTitle( const short item, Str255 title );
```

Sets the item's title to the string passed. Only some items, such as buttons, radios and checkboxes have titles.

```
virtual void              SelectItem( const short item );
```

Makes the item passed become the current focus of the dialog. This is the method to call if you wish to change this programmatically. Usually the user will do this by clicking or using the tab key.

```
virtual void              SelectNextFocus();
virtual void              SelectPreviousFocus();
```

These methods implement the standard response to the tab key and shift-tab key respectively. The next (or previous) available focusable item takes the focus.

```
virtual short             CountFocusableItems();
```

Returns the number of focusable items in the dialog.

```
virtual ZDialogItem*      GetItemObject( const short item );
```

Returns the actual ZDialogItem object of the item.

```
virtual ZDialogItem*      GetDefaultItemObject();
```

Returns the default item object (usually the "OK" button), or NULL if no default is set.

```
virtual ZCommander*       GetHandler();
```

Returns the current focussed item, or the dialog itself if there isn't one. This is the standard method used to set up the command chain- the focused item begins the command chain.

```
virtual void              AppendItemsToDialog( const short ditlID,
                                               DITLMethod apMethod = overlayDITL );
```

This method permits you to extend the dialog by appending a further set of items from a 'DITL' resource. This sees to it that the relevant objects are set up correctly. <apMethod> determines how the additional items are overlaid- refer to Inside Macintosh (this value is passed on to AppendDITL). You can call this more than once to add a series of overlays.

```
virtual void              RemoveAppendedItems();
```

Removes all appended items made with the above method, but does not remove the original items. This should only be called once, even if more than one set of items was added- all added items will be removed in one go.

```
inline  short             GetBaseItemCount();
```

Returns the number of items in the original dialog, before any items were appended. If you have appended items, you can use this to help you calculate your item numbers for the appended items, which are renumbered by adding this value.

```
virtual void              WriteToStream( ZStream* aStream );
virtual void              ReadFromStream( ZStream* aStream );
```

Writes and Reads the dialog to the stream.

```
virtual Boolean           AcceptsFlavour( const OSType aFlavour );
```

Overrides the ZWindow method- by default it returns FALSE, rejecting all drag/drops to the dialog. If your dialog can hande drops, you will need to overridethis.

```
inline  Boolean           IsModal();
```

Retuns TRUE if the dialog is modal (i.e. blocks the command chain).

```
virtual Boolean           RunModal();
```

This method handles all user interaction with a modal dialog, running it “inline” (see ZDialog chapter in User Manual). It also initialises and shows the dialog if you have not already done so, so can be the only line of code you need! It returns TRUE if the dialog was dismissed using the default item, otherwise FALSE. Note that it is the caller’s responsibility to delete the dialog object when running it in this manner.

```
virtual void              DismissModal( const short itemDismiss );
```

To force an inline modal dialog to exit gracefully, you should call this method, passing the item that should close the dialog.

```
virtual void              WindowResized();
```

This method repositions the dialog items after the dialog window size changes. The items’ sizing behaviours are set up using either an ‘ILIM’ resource or by setting them programmatically (see ZDialogItem).

```
virtual void              SetItemSizing( const short item,
                                         unsigned char sizing );
```

Sets the sizing behaviour for the item. The <sizing> parameter is a set of flags, and is most easily set up using the AUTOSIZE macro. See ZDialogItem for more details.

```
inline    short           GetDITLID();
inline    short           GetDLOGID();
```

Return the dialog’s DITL resource ID and its DLOG resource ID respectively.

```
virtual void              MakeMacWindow( const short dialogID );
```

Creates the Mac OS toolbox dialog structures when the dialog is built.The default method uses

the Dialog Manager GetNewDialog function to do most of the work. The other variant of this method is used when creating a dialog from a stream.

```
virtual void              ParseRButtonTitle(   Str255 buttonTitle,
                                               short* groupID,
                                               Boolean* isDefault );
```

Internal utility method reads the group ID from the radio button title string passed in, returning the group ID and whther the “default” asterisk was present. The string itself is modified to remove the meta-characters.

```
virtual void              HandleRButtonGroupClick( const short item );
```

Internal method to deal with radio button groups. The other members of the group are located and turned off, and the passed item is turned on.

```
virtual void              ClearDITLPlaceHolders( Handle ditl );
```

Internal method used solely by the streaming method. This modifies the DITL handle in a way necessary for the well-being of the Dialog Manager. Do not use or override.

```
virtual Boolean           PasteDataIsLegal( const short targetItem );
```

This method checks whether the current data on the clipboard could be legally pasted into the item. Called as part of the standard Edit menu maintenance. The item iself determines whether the data is legal.

```
virtual void              ParseEditFieldInfo( Str255 efText,
                                              unsigned short* efFlags,
                                              long* min, long* max );
```

Internal method reads the meta-data associated with the original text string of an edit field. This extracts the flags and the min and max value of the field, and modifies the string to remove the extra data.

```
virtual ZDialogItem*      MakeItemObject( const short item,
                                          const long magicType = 0 );
```

Creates theactual ZDialogItem (or subclass) for the item, given its “magic” or extended type. If you define additional types of dialog item objects, you need to override or modify this method so that your object is built on demand. The standard method knows how to make all currently shipping dialog item objects- however, you need to switch _DIALOG_EXTENSIONS ON in order to create the more esoteric custom types.

```
virtual void              BuildDialogObjects( short fromItemNo = 1,
                                              const short ictbID = 0 );
```

Internal method creates the required dialog item objects by converting the original Dialog Manager DITL information. This is also used when appending items, in which case <fromItemNo> tells it where to start working from. <ictbID> is the resorce ID of an ‘ictb’ resource to use to set up the initial font and colour information for the items.

```
virtual void              ParseStatText( Str255 sText,
                                         OSType* typeParam,
                                         short* paramCount,
                                         long params[] );
```

Internal method to extract the “magic” parameter list from special static text items. The original text is passed in <sText>, and if the string is suitably formatted (to qualify, must start with ‘$$’ followed by a four-letter code) it returns the four-letter code converted to an OSType, the parameter count, and the parameters themselves. Up to ten parameters can be defined- most items use far fewer.

```
virtual void              ClipOutItemsBelow( const short item );
```

This method removes the bounds rectangles of all of the items with a number lower than <item> from the window’s clip region. This is done when drawing the group boxes, so overlaid text labels are not drawn through.

```
virtual void              UserInitialise( ZDialogItem* theItem );
```

This method is provided so your dialog subclass can do any special initialisation of its own on a per-item basis. Rarely required.

```
virtual   long            DMToMagicType( short dmType );
```

Returns the “magic” type code for the standard Dialog Manager items. Since these standard items don’t use our usual “magic” static-text method, this method fills in for them.

```
virtual void              InitItemFromICTB(    const short item,
                                               ictbHandle ictb,
                                               const short ictbIndex );
```

Internal method sets up the initial font and colour information for a dialog item from an ‘ictb’ resource whose handle is passed in, together with an index into the ictb. (Due to the possibility of appended items, this value may not be the same as the item number). Do not use directly.

```
virtual void              InitItemSizingFromRes( const short resID );
```

Internal method sets up the automatic item positioning from a ‘ILIM’ resource with the ID passed. This resource should have the same ID as the DITL resource for the dialog.

Comments

Dialogs can be complex. There is quite a lot of confusion about whether it is the item or the dialog itself that should handle certain things- sometimes they work in tandem. To use dialogs effectively, please read and understand the ZDialog chapter in the User Manual. If in doubt, the final say is in the source code- by following this, it should clarify any confusing points. In general, ZDialogItems are designed in isolation- they know nothing except themselves and the dialog. The dialog is designed to deal with sets of items. However, because the dialog is usually the object that communicates with its owner in the application, it contains many helpful methods which appear to blur this distinction.

Dialog Messages

ZDialog transmits a number of messages usingthe ZComrade mechanism. Usually, this is how the dialog's owner will be able to respond to the dialog and extract data when it closes. The messages are:

```
enum
{
     kMsgDialogSuccessfullyClosed    = 'dlg$',
     kMsgDialogCancelled             = 'dlg-',
     kMsgDialogItemClicked           = 'dlg~',
     kMsgDialogSetUp                 = 'dlg!'
};
```

"Magic" Types

Currently defined extended type codes for all items are as follows:

```
enum
{
     kMagicStringListbox             = 'LIST',       // implemented
     kMagicStringScrollbox           = 'TEXT',       // implemented
     kMagicStringIconList            = 'ICLB',       // implemented
     kMagicStringColourMenu          = 'CPOP',       // implemented
     kMagicStringProgressBar         = 'PBAR',       // implemented
     kMagicStringCounterDisplay      = 'DIGI',       // not yet implemented, reserved
     kMagicStringClockDisplay        = 'CLOC',       // not yet implemented, reserved
     kMagicStringGWorldScroller      = 'GWRP',       // implemented
     kMagicStringGenericScroller     = 'SCRL',       // implemented
//-----------------------------------------
     kMagicStdStaticText             = 'stat',
     kMagicStdEditText               = 'edit',
     kMagicStdPushButton             = 'butt',
     kMagicStdCheckbox               = 'chek',
     kMagicStdRadioButton            = 'radi',
     kMagicStdResControl             = 'cntl',
     kMagicStdIcon                   = 'icon',
     kMagicStdPicture                = 'pict',
     kMagicStdUserItem               = 'user'
};
```

Class Name: ZDialogItem	Based on: ZCommander
Type:	Framework Class
Description:	Generic dialog item handles standard items. Can be subclassed to implement custom dialog items.

ZDialogItem is the basis for all dialog items. ZDialogItem itself handles all of the standard dialog items such as buttons, chackboxes, edit text, etc. It is subclassed to implement custom dialog items such as list boxes and so on.

Class Definition

```
class     ZDialogItem     : public ZCommander
{
protected:
     short               iType;          // item type flags
     short               id;             // id number in dialog
     short               groupID;        // group ID number or zero if not in a group
     unsigned short      efFilterFlags;  // edit field capability flags
     unsigned long       hHistory;       // state history stack
     long                maxLim;         // max value
     long                minLim;         // min value
     long                xType;          // extended type code
     Rect                bounds;         // bounds rect
     short               font;           // font for item
     short               fontSize;       // font size for item
     Style               fontStyle;      // font style for item
     Boolean             enabled;        // item is available for clicks
     Boolean             focused;        // item has keyboard focus
     Boolean             canTakeFocus;   // item eligible for kb focus
     Boolean             hilited;        // item is not disabled
     Boolean             visible;        // item is visible in window
     Boolean             isDefault;      // item maps to return/enter keys
     Sizing              fSizing;        // autosizing of item bounds
     Handle              macItemHandle;  // handle to dialog item
     TEHandle            pwMirror;       // text edit record for password
     RGBColor            bkColour;       // background colour of item
     RGBColor            fgColour;       // foreground colour of item

public:

     ZDialogItem();
     ZDialogItem( ZDialog* aBoss, const short anID );
     ~ZDialogItem();

// initialisation
     virtual void        InitItem( const short paramCount, const long params[] );

// standard commander type stuff
     virtual void        Idle();
     virtual void        Type( const char theKey, const short modifiers );
     virtual void        UpdateMenus();
     virtual void        BecomeHandler( Boolean isBecoming );

// appearance
     virtual void        Draw();
     virtual void        Enable( Boolean useHistory = FALSE );
     virtual void        Disable( Boolean useHistory = FALSE );
```

```
      virtual void         Show();
      virtual void         Hide();
      virtual void         DoHilightSelection( Boolean hiliteIt );
      virtual void         DrawDefaultOutline();

// mousing around
      virtual void         Click( const Point mouse, const short modifiers );
      virtual void         AdjustCursor( const Point mouse, const short modifiers );
      virtual Boolean      GetBalloonHelp( const Point mouseIn,
                                           Rect* rectOut,
                                           Point* tipOut,
                                           HMMessageRecord* hm );


// edit menu handling
      virtual Boolean      CanPasteType();
      virtual void         DoCut();
      virtual void         DoCopy();
      virtual void         DoPaste();
      virtual void         DoClear();
      virtual void         DoSelectAll();

// setting the value
      virtual void         SetValue( const long aValue );
      virtual void         SetValue( const int value );
      virtual void         SetValue( const short value );
      virtual void         SetValue( const Str255 value );
      virtual void         SetValue( const double value );
      virtual void         SetValue( const float value );

// getting the value
      virtual long         GetValue();
      virtual void         GetValueAsText( Str255 aStr );
      virtual float        GetValueAsFloat();

// limits and checks
      virtual void         SetMaximum( const long max );
      virtual void         SetMinimum( const long min );
      virtual void         GetLimits( long* min, long* max );
      virtual Boolean      Validate();
      virtual Boolean      CheckKey( char* theKey );

// information
      inline     Boolean    IsEnabled();
      inline     Boolean    IsVisible();
      inline     Boolean    HasKeyboardFocus();
      inline     Boolean    CanTakeKeyboardFocus();
      inline     void       SetCanTakeFocus( Boolean canTake );
      inline     Boolean    IsDefaultItem();
      inline     void       SetDefaultItem( Boolean is );
      inline     short      GetID();
      inline     short      GetGroupID();
      inline     void       SetGroupID( const short id );
      inline     void       SetFilterFlags( const short flags );
      inline     unsigned short  GetFilterFlags() { return efFilterFlags; };
      inline     short      GetType();
      inline     void       GetBounds( Rect* aRect );
      inline     void       SetXType( long aType );
      inline     long       GetXType();
      inline     Handle     GetMacItemHandle();

      virtual WindowPtr     GetMacDialog();

// font and colours
      virtual void         SetFontInfo( const short fontID, const short size,
                                        const Style style );
      virtual void         SetBackColour( RGBColor* aColour );
```

```
     virtual void         SetForeColour( RGBColor* aColour );

// titling
     virtual void         SetTitle( Str255 aTitle );
     virtual void         GetTitle( Str255 aTitle );

     virtual void         GetDebugInfoString( Str255 s );

// item sizing
     virtual void         SetBounds( Rect* newBounds );
     virtual void         ParentResized( Rect* oldParentBounds,
                                         Rect* newParentBounds );
     virtual void         SetAutoSizing( Sizing sizeOptions );

protected:

// private stuff
     virtual void         PrepareForDrawing();
     virtual Boolean      PopStateHistory();
     virtual void         PushStateHistory( Boolean aState );
     virtual void         SubstituteParamText( Str255 aString );
     virtual void         DrawStdFrame( Rect* frame );
     virtual void         DrawFocusBorder( Boolean bState );

// drawing methods for standard item types
     virtual void         DrawItem();
     virtual void         DrawUserItem();
     virtual void         DrawPictureItem();
     virtual void         DrawIconItem();
     virtual void         DrawControlItem();
     virtual void         DrawTextItem();

     void                 ValidItem();
     void                 InvalItem();
     void                 FocusBoss();
};
```

Data Members

<**iType**> the original Dialog Manager type of the item, less thedisabled flag
<**id**> the item number of this item
<**groupID**> the group ID this item belongs to, or 0 if none
<**efFilterFlags**> the feature flags for an edit field
<**hHistory**> the enable state history
<**maxLim**> the maximum value limit for an edit field
<**minLim**> the minium value limit for an edit field
<**xType**>the extended “magic” type of the item
<**bounds**> the bounds rectangle ofthe item in the dialog window
<**font**> the font ID to draw text with
<**fontSize**> the font size of the item’s text
<**fontStyle**> the style of the item’s text
<**enabled**> TRUE if the item will respond to mouse clicks
<**focused**> TRUE ifthe item has the user’s focus
<**canTakeFocus**> TRUE if the item is able to take the focus
<**hilited**> TRUE if the item is enabled (not dimmed)
<**visible**> TRUE if the item is actually drawn
<**isDefault**> TRUE if this is thedefault item for the dialog
<**fSizing**> auto repositioning flags for the item
<**macItemHandle**> the handle to the Mac OS obejct for the item- e.g. ControlHandle, etc.

<**pwMirror**> TextEdit record for password handling
<**bkColour**> the background colour of the item
<**fgColour**> the foreground colour of theitem

Methods

```
ZDialogItem();
ZDialogItem( ZDialog* aBoss, const short anID );
~ZDialogItem();
```

Constructors create the basic object, the parameterless version being for streaming. The item's boss is always the dialog that contains it. The data memebers are initialised to their default values, and some basic information is read from the Dialog Manager using the GetDialogItem function.

```
virtual void        InitItem( const short paramCount, const long params[] );
```

Performs the rest of the initialisation for the item. Any parameters that were set up as part of the "magic" string of a static text item are passed in here. Up to ten parameters can be passed per item. Note that unused parameters are set to 0, and only parameters from 0 to paramCount -1 need to be read. Parameters of value 0 should be treated as the default value for that setting. The individual assignments of each parameter are determined by the indiviual item. For standard items, the parameters are not used.

```
virtual void        Idle();
```

Called repeatedly while the item has the focus. Used to blink the caret in edit fields, etc.

```
virtual void        Type( const char theKey, const short modifiers );
```

Respond to the keyboard. Edit fields pass characters to TextEdit. Dialog keys such as tab, return, enter and escape are passed back to the dialog for processing.

```
virtual void        UpdateMenus();
```

Enable commands the item can respond to. This usually includes Cut, Copy, Paste, etc.

```
virtual void        BecomeHandler( Boolean isBecoming );
```

Called by the dialog when the focus is changing. The item which had the focus is called with <isBecoming> set to FALSE, and the item that is gaining the focus is called with <isBecoming> set to TRUE.The standard method, which will usually suffice, responds by drawing or erasing the focus ring and calling DoHiliteSelection().

```
virtual void        Draw();
```

Basic drawing method. You shouldn't override this- override DrawItem() instead. This sets up the drawing port with the item's font and colours, then calls DrawItem.

```
virtual void        Enable( Boolean useHistory = FALSE );
virtual void        Disable( Boolean useHistory = FALSE );
```

Enables and Disables the item, optionally recording the state history. The item responds by

redrawing th eitem as required. Note- this works for edit fields also- something the Dialog Manager is unable to do (though new Appearance controls add this functionality).

```
virtual void         Show();
virtual void         Hide();
```

Shows and Hides the item

```
virtual void         DoHilightSelection( Boolean hiliteIt );
```

Highlights the item as appropriate to reflect whether it has the focus or not.
Items that do not have the focus should usually deactivate their selections. There are exceptions to this, so this must be considered on an item-by-item basis. The default method handles the highlighting of TextEdit fields. This is commonly overridden in subclasses.

```
virtual void         DrawDefaultOutline();
```

Draw the outline of the item if it is the default item. The default method does nothing unless the item is a standard button, in which case it draws the usual additional border.

```
virtual void         Click( const Point mouse, const short modifiers );
```

This method responds to the mouse click in the item. For standard items, the usual response is to track a control or position the caret in an edit field. If the item does not have the focus but may, the item is first focussed by calling ZDialog's SelectItem() method. Note- items that do not have the "enabled" flag set in the original DITL will not respond to the mouse. This should not be confused with whether the item is visibly dimmed or enabled.

```
virtual void         AdjustCursor( const Point mouse, const short modifiers );
```

Set the cursor shape as appropriate for the item. Edit fields set the iBeam cursor.

```
virtual Boolean      GetBalloonHelp( const Point mouseIn,
                                     Rect* rectOut,
                                     Point* tipOut,
                                     HMMessageRecord* hm );
```

Returns standard balloon help information for the item. The default method looks up the help message in a 'hdlg' resource with the same ID as the DITL, correctly setting up the enabled and checked state for the messageas required. You can overridethis to obtain help in another way if you prefer.

```
virtual Boolean      CanPasteType();
virtual void         DoCut();
virtual void         DoCopy();
virtual void         DoPaste();
virtual void         DoClear();
virtual void         DoSelectAll();
```

Standard Edit menu handling. These commands work for edit fields as expected. CanPasteType() checks that the clipboard contains data of type 'TEXT'.

```
virtual void         SetValue( const long aValue );
virtual void         SetValue( const int value );
virtual void         SetValue( const short value );
virtual void         SetValue( const Str255 value );
virtual void         SetValue( const double value );
virtual void         SetValue( const float value );
```

SetValue sets the item's value to that passed. You can pass data of any type and the item will do the expected thing, applying any necessary conversions. This is the usual way to set up the data state of all dialog items.

```
virtual long         GetValue();
virtual void         GetValueAsText( Str255 aStr );
virtual float        GetValueAsFloat();
```

Returns the value of the item, as the data type requested. Any necessary conversions are done. This is the normal way to get the data state of an item.

```
virtual void         SetMaximum( const long max );
virtual void         SetMinimum( const long min );
virtual void         GetLimits( long* min, long* max );
```

For an edit text field,this allows you to set up the limits for the numerical value of the item. GetLimits returns the current limits. For your own items, you can define what these limits are used for as you wish.

```
virtual Boolean      Validate();
```

Validates the item. For edit fields that havethe limit flag set, the numerical value of the data in the field is checked against the set min and max. If the value falls outsidethe range, this returns FALSE, if OK, it returns TRUE. For your own items, you can validat ethe item as you wish. This is called when the dialog closes to verify that the user entered sensible data. If you return FALSE, dialog closureis prevented.

```
virtual Boolean      CheckKey( char* theKey );
```

Determines if the typed character may be entered into the item. Edit fields may have flags set that reject certain characters to help the user enter sensible data, e.g. numeric only. This returns FALSE is the character is bad, otherwise TRUE.

```
inline    Boolean    IsEnabled();
```

Returns whether the item is hilited (i.e. not dimmed). This should not be confused with the original DITL "enabled" flag which merely sets if the item should respond to the mouse.

```
inline    Boolean    IsVisible();
```

Returns TRUE if the item is visible.

```
inline    Boolean    HasKeyboardFocus();
```

Returns TRUE if the item currently has the focus.

```
inline    Boolean    CanTakeKeyboardFocus();
```

Returns TRUE if the item could potentially take the keyboard focus, whether or not it currently has.

```
inline    void      SetCanTakeFocus( Boolean canTake );
```

Sets whether the item is eligible to take the focus. You should rarely need this- your items can simply set the ‘canTakeFocus’ data member directly when they are created.

```
inline    Boolean   IsDefaultItem();
```

Returns TRUE if the item is the default item for the dialog.

```
inline    void      SetDefaultItem( Boolean is );
```

Sets the item as the default item (or not) this is called as part of the initialisation by the dialog. Do not use it directly, this is set up by ZDialog.

```
inline    short     GetID();
```

Returns the item’s ID number.

```
inline    short     GetGroupID();
inline    void      SetGroupID( const short id );
```

Returns and sets the group ID for the item. usually this is set up by the dialog (e.g. for radio buttons) but you can programmatically set other items as part of a group for dimming purposes, etc.

```
inline    void      SetFilterFlags( const short flags );
inline    unsigned short  GetFilterFlags();
```

Sets the filter flags for an edit field, and returns those flags. Normally this is done by the dialog as part of the initialisation for an edit field.

```
inline    short     GetType();
```

Returns the original Dialog Manager type of the item. Use with care- you may not get what you were expecting! An item may “fake” the type to take advantage of certain standard processing. For example, the item may not be a control, but as long as it sets <macItemHandle> to a valid ControlHandle and sets the ctrl bit if the item type, it will get standard control processing for free (such as hiliting). Custom dialog items should also set the top bit of the type to indicate that they are a non-native item.

```
inline    void      GetBounds( Rect* aRect );
```

Returns the bounds rectangle of the item.

```
inline    void      SetXType( long aType );
inline    long      GetXType();
```

Sets and returns the extended type code for the item. This is more reliable than the original Dialog Manager type since no fakery can occur! You should not set the X type- the dialog does this.

```
inline    Handle    GetMacItemHandle();
```

Returns the original Dialog Manager handle for the item's data, if any. For example, for a control item, this is the ControlHandle of the control. Use with great care! Custom Items may use this handle for their own purposes, but must be very careful to prevent the Dialog Manager doing something stupid with it- the Dialog Manager will believe the values and flags set in the iType field, and treat the handle accordingly. While this can be to your advantage, you must take care not to mislead the Dialog Manager, because it's almost certain it will crash your program.

```
virtual WindowPtr    GetMacDialog();
```

Returns the original Mac dialog as a WindowPtr. Rarely required.

```
virtual void         SetFontInfo( const short fontID, const short size,
                                  const Style style );
```

Sets the font information for the item. Subsequent drawing will draw using these parameters. If you pass 0, the dialog's font will be used. This is often set up automatically using an 'actb' resource, but you can call it yourself to set the font info programmatically if you prefer. Note that this does NOT affect the font used when drawing buttons and controls- they always draw in the system font due to the way that controls are handled in the Dialog Manager. A workaround is to create controls using a 'CNTL' resource and set the useWFont bit in the procID. The font set here will then be used when the button or control is drawn.

```
virtual void         SetBackColour( RGBColor* aColour );
virtual void         SetForeColour( RGBColor* aColour );
```

Sets thebackground and forgeround colours for drawing the item. Usually set up from an 'ictb'resource, but you can call them yourself as needed.

```
virtual void         SetTitle( Str255 aTitle );
virtual void         GetTitle( Str255 aTitle );
```

Sets and returns the title of the item. Only some kinds of items have titles- namely controls. Custom items may have titles as well.

```
virtual void         GetDebugInfoString( Str255 s );
```

Returns readable information about the item that can be used by debuggers and inspectors.

```
virtual void         SetBounds( Rect* newBounds );
```

Moves the item to the new size and position passed.

```
virtual void         ParentResized( Rect* oldParentBounds,
                                    Rect* newParentBounds );
```

Called by the dialog when the size of the window changes. This responds by working out from the sizing flags set for the item what the new bounds should be and calling SetBounds to reposition the item.

```
virtual void         SetAutoSizing( Sizing sizeOptions );
```

Sets the sizing options for the item. Each side of the item's bounds can move so it maintains its

relative distance from either edge of the dialog, or the relative proportion between the two. The easiest way to set this up is using the AUTOSIZE macro. See the comments section for more information.

```
virtual void         PrepareForDrawing();
```

Sets up the current port with the font and colour information of the item, prior to doing any other drawing. This is called by Draw and other methods that redraw the item. You should use a ZGrafState object to preserve the port state prior to making this call.

```
virtual Boolean      PopStateHistory();
virtual void         PushStateHistory( Boolean aState );
```

Pushes and Pops the items hilited state history. This allows Enable and Disable calls to be nested to give the right behaviour. The nesting may be up to 32 levels deep, but would rarely be more than one or two. Do not call directly- these are called by Enable and Disable.

```
virtual void         SubstituteParamText( Str255 aString );
```

If your item can use the standard parameter text mechanism (as static and editable text can) this performs the substitution. Your own items can take advantage of this if they wish. The standard parameter text as set using the ParamText function is substituted for ^0, ^1, ^2 and ^3 in the string passed.

```
virtual void         DrawStdFrame( Rect* frame );
```

Draws a standard frame around the item. This consists of a black (or grey, if dimmed) border 1 pixel OUTSIDE the bounds rect of the item, plus a grey 3D effect border outside of that (drawn using FrameGrayRect). This method is used when NOT using appearance. You can call it for your own items as needed.

```
virtual void         DrawFocusBorder( Boolean bState );
```

Draws a standard focus ring around the item. This method is appearance aware, using the appearance focus ring if possible. If not, it mimics the standard appearance using a dark gray colour. This is called as necessary by BecomeHandler().

```
virtual void         DrawItem();
```

Draws the contents of the item. The font and colours, etc are all set up. You will normally override this method for your custom items. The standard methods calls one of the following methods according to the item type:

```
virtual void         DrawUserItem();
virtual void         DrawPictureItem();
virtual void         DrawIconItem();
virtual void         DrawControlItem();
virtual void         DrawTextItem();
```

Draws the standard types of items. In addition, extra info such as the hilited state is honoured for all items, so yes, you can grey out icons simply by calling the Disable() method on them... take that, Dialog Manager!

```
	void		ValidItem();
	void		InvalItem();
	void		FocusBoss();
```

These utility methods are handy for your own items. ValidItem() removes the item's bounds from the update region, if you've already drawn it, for example. InvalItem() add the boubds to the update region- the item will be redrawn at the next update. FocusBoss() makes the owning dialog the current port.

Comments

ZDialogItem looks large and cumbersome, but actually it's not so bad- it looks worse than it is because the standard item handles all of the normal types of item in one object. Your own custom dialog items will usually be much simpler. Generally, you may be able to get away with only overriding a small number of methods, such as DrawItem(), Click() and DoHilightSelection(). You will inherit a lot of standard behaviour from the base item. As with many parts of MacZoop, if you find yourself doing too much, you're probably doing it wrong. If you are intending to create a custom dialog item, it's worth studying one of the pre-supplied ones first to see how they do it.

Automatic item sizing

To take advantage of automatic item sizing, you can either supply a 'ILIM' resource which sets up the whole dialog in one go, or else call each item's SetAutoSizing() method individually. In either case, the <sizingOption> parameter must be set up the same way. Each side of the item is represented by two bits, giving four options per side. Four sides gives you 4 x 2 = 8 bits, a char. The bits are defined thus:

```
enum
{
	NONE		= 0,
	FIXEDLEFT	= 1,
	FIXEDTOP	= 1,
	FIXEDRIGHT	= 2,
	FIXEDBOTTOM	= 2,
	PROPORTIONAL	= 3
};
```

And must be shifted into place for the four sides. The easiest way is to use the AUTOSIZE macro:

```
AUTOSIZE( t, l, b, r );
```

where t, l b and r are the top, left, bottom and right sides of the item. e.g.

```
sizing = AUTOSIZE( FIXEDTOP, FIXEDLEFT, FIXEDBOTTOM, PROPORTIONAL );
item->SetAutoSizing( sizing );
```

The ILIM resource is simply a list of such sizing bytes, together with the ID of the item to apply it to (that way your ILIM can set some items but not others). The dialog will load and read any ILIM resource, which should have the same ID as the DITL resource that specified the items.

Class Name: ZScroller	Based on: ZWindow
Type:	Framework Class
Description:	Window object that implements scrollingin a standard way. Most real windows will be based on this.

ZScroller is based on ZWindow and can do everything it can. In addition, it supports a general method for scrolling which allows you to set this up very easily. Most real-world document interfaces are likely to be based on ZScroller rather than ZWindow. As well as providing a virtual drawing area as large as the QuickDraw plane, this also provides standard methods for adding a header and margin to your window.

Class Definition

```
class ZScroller : public ZWindow
{
protected:
     ControlHandle        theHBar;        // handle to horizontal scrollbar
     ControlHandle        theVBar;        // handle to vertical scrollbar
     Rect                 bounds;         // scrollable area (virtual document size)
     short                hScale;         // pixels shifted per unit horizontally
     short                vScale;         // pixels shifted per until vertically
     Boolean              hasHBar;        // true if has horizontal bar
     Boolean              hasVBar;        // true if has vertical bar
     short                cInitValue;     // used for live scrolling support
     short                topMargin;      // margin for header area
     short                leftMargin;     // width of margin on left

public:

     ZScroller( ZCommander* aBoss,
                    const short windID,
                    const Boolean hasHScroll = TRUE,
                    const Boolean hasVScroll = TRUE );

     ZScroller();

// ZWindow overrides:
     virtual void    InitZWindow();
     virtual void    Activate();
     virtual void    Deactivate();
     virtual void    Draw();
     virtual void    DrawGrow();
     virtual void    Click( const Point mouse, const short modifiers );
     virtual void    SetSize( const short width, const short height,
                              const Boolean reDraw = TRUE);
     virtual void    Zoom( const short partCode );
     virtual void    SetBounds( const Rect& aBounds );
     virtual void    SetBounds( short top, short left, short bottom, short right );
     virtual void    SetBounds( Point tl, Point br );
     virtual void    GetBounds( Rect* aBounds );
     virtual void    GetIdealWindowZoomSize( Rect* idealSize );

     virtual void    GetContentRect( Rect* aRect );
     virtual void    ClickContent( const Point mouse, const short modifiers );
     virtual void    ClickScroll( const Point mouse );
     virtual void    AdjustCursor( const Point mouse, const short modifiers );
```

```
// ZScroller methods:
     virtual void    SetScrollAmount( const short hAmount, const short vAmount );
     virtual void    GetPosition( short* hPosition, short* vPosition );
     virtual void    ScrollTo( const short hPosition, const short vPosition );
     virtual void    Scroll( const short dH, const short dV );

     virtual Boolean AutoScroll( Point mousePt );
     virtual Boolean WillScroll( Point mousePt );
     virtual void    SetOriginToScroll();

// margins:
     virtual void    DrawHeader();
     virtual void    DrawLeftMargin();
     virtual void    ClickHeader( const Point mouse, const short modifiers ) {};
     virtual void    ClickLeftMargin( const Point mouse, const short modifiers ) {};

// streaming:

     virtual void    WriteToStream( ZStream* aStream );
     virtual void    ReadFromStream( ZStream* aStream );

protected:

     virtual void    CalculateControlParams();
     virtual void    MakeScrollbars();
     virtual void    MoveScrollbars();
     virtual void    PostScroll( ControlHandle aCtl = NULL );
     virtual void    HideScrollbars( Boolean validateArea = FALSE );

public:

     virtual void    ScrollHandler( const ControlHandle aCtl, const short partCode );
};
```

Data Members

<**theHbar**> and <**theVBar**> are the handles to the scrollbar controls.
<**bounds**> is the rectangle of the scrollable area (virtual drawing space).
<**hScale**> and <**vScale**> are the amount to shift the view for each click in the scroll arrows.
<**hasHBar**> and <**hasVBar**> specify which scrollbars should be added to the window.
<**cInitValue**> is used to track the original position when live scrolling.
<**topMargin**> is the height of the window header area.
<**leftMargin**> is the width of the window left margin area.

Methods

```
     ZScroller( ZCommander* aBoss,
                     const short windID,
                     const Boolean hasHScroll = TRUE,
                     const Boolean hasVScroll = TRUE );
```

Constructor sets up the basic object. As well as the usual boss and window ID parameters, you can pass which scrollbars your window should have. Usually it will be both (the default).

```
     virtual void    InitZWindow();
```

Creates the window by calling the inherited method, then creates the actual scrollbars.

```
     virtual void    Activate();
     virtual void    Deactivate();
```

Overrides the ZWindow methods to additionally decativate and reactivate the scrollbars.

```
virtual void    Draw();
```

Overrides the ZWindow method to take additional action when setting up for drawing. This includes offsetting the port to allow for the scroll position, setting the clip region, and updating the scrollbars.

```
virtual void    DrawGrow();
```

Draws the grow box.

```
virtual void    Click( const Point mouse, const short modifiers );
```

Overrides the ZWindow method to handle clicks in the scrollbars themselves, which scroll the view as required. Clicks in the content area are passed to a new method, ClickContent(), which is what you should override to process mouse clicks in the drawing area.

```
virtual void    SetSize( const short width, const short height,
                         const Boolean reDraw = TRUE);
```

Overiddes ZWindow to reposition the scrollbars when the window changes size. Calls the original method to actually change the size.

```
virtual void    Zoom( const short partCode );
```

Likewise for zooming.

```
virtual void    SetBounds( const Rect& aBounds );
virtual void    SetBounds( short top, short left, short bottom, short right );
virtual void    SetBounds( Point tl, Point br );
```

Sets the bounds rectangle. This is the virtual drawing area, which may be much larger than the window. The full range of the QuickDraw plane is available from -32,768 to 32,767 both horizontally and vertically. SetBounds may be called at any time, and will recalculate the scrollbar settings as needed.

```
virtual void    GetBounds( Rect* aBounds );
```

Returns the bounds rectangle.

```
virtual void    GetIdealWindowZoomSize( Rect* idealSize );
```

Returns the ideal window size to the Zoom() function. By default, this is equal to the bounds.

```
virtual void    GetContentRect( Rect* aRect );
```

Returns the interior part of the window, less the scrollbar and margin areas.

```
virtual void    ClickContent( const Point mouse, const short modifiers );
```

Your opportnity to respond to mouse clicks in the drawable part of the window. The default method installs a mouse tracker (if this feature enabled). You should override it if you want to

process the mouse click in your view.

```
virtual void    ClickScroll( const Point mouse );
```

This method drags the scrollable view around with the mouse, implementing the “grabber” type tool. It is called usually when the user command-clicks in the window.

```
virtual void    AdjustCursor( const Point mouse, const short modifiers );
```

Sets the cursor shape according to which part of the window the mouse is over. The default method sets the “grabber” hand cursor if the mouse is over the interior area and the command key is down. Override for other behaviour.

```
virtual void    SetScrollAmount( const short hAmount, const short vAmount );
```

Sets the amount the view will scroll for each single click in the scroll arrow of the scrollbar. Typically, this will be 8 or 10 pixels, or the lineheight of text in the view, etc.

```
virtual void    GetPosition( short* hPosition, short* vPosition );
```

Returns the current scroll position of the view. This is the position of the top, left corner of the interior of the window with respect to the bounds rectangle.

```
virtual void    ScrollTo( const short hPosition, const short vPosition );
```

Moves the view to the positions shown. Can be used to programmatically scroll the view.

```
virtual void    Scroll( const short dH, const short dV );
```

Scrolls the view by <dH> pixels horizontally and <dV> pixels vertically, redrawing any newly revealed area. You would not normally use this method directly, but if you do, be careful to set up the Focus and scroll offset before and after.

```
virtual Boolean AutoScroll( Point mousePt );
```

Autoscroll the view if necessary, based on the mouse point passed. This is usually called within some kind of drag tracking loop to auto-scroll the view. If the view actually scrolled, this returns TRUE, otherwise FALSE. The view will scroll if <mousePt> is outside of the content rectangle, and the distance scrolled will depend on how far outside the mouse is.

```
virtual Boolean WillScroll( Point mousePt );
```

Can be used to preflight an auto-scroll call in certain circumstances. It returns TRUE if the view would scroll if AutoScroll was called with the samepoint, otherwise FALSE.

```
virtual void    SetOriginToScroll();
```

Sets up the origin of the port and the clip region to allow for a scroll. Called by Draw() and other methods. You ned to call this if you are drawing in your view and were not called through the usual channels.

```
virtual void    DrawHeader();
virtual void    DrawLeftMargin();
```

These methods draw the header and margin areas o fthe window, if the sizes set for these are non-zero. You would usually override them to actually draw what you wanted in these areas. The default methods draw these areas using Appearance panels if available.

```
virtual void    ClickHeader( const Point mouse, const short modifiers ) {};
virtual void    ClickLeftMargin( const Point mouse, const short modifiers ) {};
```

Respond to mouse clicks in the header and margin areas. The default methods do nothing- you need to override these to handle the mouse.

```
virtual void    WriteToStream( ZStream* aStream );
virtual void    ReadFromStream( ZStream* aStream );
```

Writes and Reads the scroller to the stream.

```
virtual void    CalculateControlParams();
```

Computes the correct min, max and value for the scrollbars. Called whenever the bounds or window size changes.

```
virtual void    MakeScrollbars();
```

Creates the scrollbar controls when the window is built.

```
virtual void    MoveScrollbars();
```

Moves the scrollbars to the window edges after a resize. You can override this if your scrollbars have special positioning properties- for example often the horizontal bar allows space on the left for a placard, etc. The bars are hidden prior to this call, and this must reshow them after moving them.

```
virtual void    PostScroll( ControlHandle aCtl = NULL );
```

Called after the view is scrolled using the scrollbar thumb. Rarely used.

```
virtual void    HideScrollbars( Boolean validateArea = FALSE );
```

Hides the scrollbars, optionally marking the area as valid.This is done prior to moving them when the window changes size.

```
virtual void    ScrollHandler( const ControlHandle aCtl, const short partCode );
```

Callback method handles scrollbar tracking and live scrolling (actually scrolling the view as the scrollbar is tracked). You should not use this directly.

Class Name: ZTextWindow	Based on: ZScroller
Type:	Framework Class
Description:	Standard text window based on TextEdit, handles styles text.

ZTextWindow implements a standard text window where you can read and edit text. It supports styled text and automatically manages the standard Font, Style and Size menus. Based in TextEdit, it has all the features and limitations in that toolbox manager.

Class Definition

```
class ZTextWindow     : public ZScroller
{
protected:
      TEHandle             itsText;
      Boolean              isEditable;
      short                emSpace;
      short                emWidth;
      TEWidthControl       wControl;

public:
      ZTextWindow( ZCommander* aBoss, const short windID,
                   Boolean allowEditing = TRUE );
      ZTextWindow();
      ~ZTextWindow();

      virtual void    InitZWindow();
      virtual void    DrawContent();
      virtual void    ClickContent( const Point mouse, const short modifiers);
      virtual void    Activate();
      virtual void    Deactivate();
      virtual void    SetSize( const short width, const short height);
      virtual void    Zoom( const short partCode );
      virtual void    Scroll( const short dH, const short dV );
      virtual void    Type( const char theChar, const short modifiers );
      virtual void    OpenFile( const OSType fType, Boolean isStationery = FALSE );
      virtual void    SaveFile();
      virtual void    Idle();
      virtual void    AdjustCursor( const Point mouse, const short modifiers );

      virtual Boolean CanPasteType();
      virtual void    DoCut();
      virtual void    DoCopy();
      virtual void    DoPaste();
      virtual void    DoClear();
      virtual void    DoSelectAll();

      virtual void    HandleCommand( const long aCmd );
      virtual void    HandleCommand( const short menuID, const short itemID );
      virtual void    UpdateMenus();

      virtual void    SetWidthControl( TEWidthControl aCtl, short fixWidth = 255 );
      virtual void    SetSizeRect( const Rect& szRect );
      virtual void    TextEditClickLoop();

      virtual void    GetTextViewRect( Rect* r ) { GetContentRect( r ); };
```

```
    inline    TEHandle   GetTextEditHandle();

protected:

    virtual void    MakeTextEdit();
    virtual void    RecalText();
};
```

Data Members

<**itsText**> is the TextEdit handle that the window uses to store and manage the text.
<**isEditable**> is TRUE if the text can be edited by the user, FALSE if it is read-only.
<**emSpace**> is the width of the widest character
<**emWidth**> is the number of characters displayed per line, approximately.
<**wControl**> controls how word-wrapping is done for text lines in the window

Methods

```
    ZTextWindow( ZCommander* aBoss, const short windID,
                 Boolean allowEditing = TRUE );
    ZTextWindow();
    ~ZTextWindow();
```

Constructors create the basic object. <allowEditing> sets whther the user will be able to edit the text, or whether it will be read only. Default contructor used for streaming. Destructor cleans up and releases any memory.

```
    virtual void    InitZWindow();
```

Initialises the window. It calls the inherited method to set up the base scroller window, then MakeTextEdit to make the text stuff.

```
    virtual void    DrawContent();
```

Updates the text when an update event is processed.

```
    virtual void    ClickContent( const Point mouse, const short modifiers);
```

Handles the mouse within the text area, passing on the clicks to TextEdit.

```
    virtual void    Activate();
    virtual void    Deactivate();
```

Handles the activation events, activating and deactivating TextEdit as needed.

```
    virtual void    SetSize( const short width, const short height);
```

Reflows the text as required when the window size changes.

```
    virtual void    Zoom( const short partCode );
```

As above for when the window is zoomed.

```
    virtual void    Scroll( const short dH, const short dV );
```

Overrides ZScroller to scroll the text in the view- because TextEdit handles its own scrolling, some of ZScroller's normal behaviours are actually deliberately bypassed in this class.

```
virtual void    Type( const char theChar, const short modifiers );
```

Passes keyboard input on to TextEdit.

```
virtual void    OpenFile( const OSType fType, Boolean isStationery = FALSE );
```

Opens a TEXT file into the window (installs the text in the file into TextEdit). This is a good example of how to implement OpenFile for real windows.

```
virtual void    SaveFile();
```

Saves the text in the window to a text file.

```
virtual void    Idle();
```

If the text is editable and active, this blinks the caret.

```
virtual void    AdjustCursor( const Point mouse, const short modifiers );
```

Sets the cursor shape to an iBeam over the text if editable, arrow if not.

```
virtual Boolean CanPasteType();
```

Returns TRUE if the clipboard contains 'TEXT' data and the text is editable.

```
virtual void    DoCut();
virtual void    DoCopy();
virtual void    DoPaste();
virtual void    DoClear();
```

Handles standard clipboard commands via TextEdit functions TECut, etc. Note that if the text is not editable, these methods do nothing.

```
virtual void    DoSelectAll();
```

Selects all of the text, if editable.

```
virtual void    HandleCommand( const long aCmd );
```

Handles commands that affect the text size, colour, alignment and style, passing others to ZScroller.

```
virtual void    HandleCommand( const short menuID, const short itemID );
```

Handles font choices in the standard Font menu, if present.

```
virtual void    UpdateMenus();
```

Enables the commands it can handle, if the text is editable.

```
virtual void    SetWidthControl( TEWidthControl aCtl, short fixWidth = 255 );
```

This method sets up how the object handles word-wrap. <aCtl> can be one of:

```
typedef enum
{
     teFixedWidth,
     teWindowWidth
}
TEWidthControl;
```

The wrapping control defaults to **teWindowWidth**, which means that the text will be wrapped at the right-hand window edge wherever that may be- it will vary as the user changes the window size. Thus changing the window size reflows the text. If you pass **teFixedWidth**, the text will wrap at a fixed position even if the window size changes. The parameter <fixWidth> specifies how many characters will be accommodated per line. This mode is intended for applications such as code editors which use a mono-spaced font such as monaco and have a set number of characters per line.

```
     virtual void    SetSizeRect( const Rect& szRect );
```

Overrides ZWindow to modify the size rect when fixed width is set so that the user can drag the window to any size.

```
     virtual void    TextEditClickLoop();
```

Standard method for implementing the TextEdit click loop. It preforms autoscrolling. You can overridethis if you want a different click loop behaviour, which is easier than installing a TextEdit clikLoop proc.

```
     virtual void    GetTextViewRect( Rect* r );
```

Returns the view rect of the text. By default this is the same as the content rect of the window.

```
     inline     TEHandle   GetTextEditHandle();
```

Returns the internal TextEdit handle.

```
     virtual void    MakeTextEdit();
```

Creates theTextEdit record and sets it up.

```
     virtual void    RecalText();
```

Rcalculates the text and scrollbar values when the window resizes or text is edited.

Comments

If your project uses this class, you should also add TextStyleUtils.cpp to the project. This code implements some standard utilities which are used to maintain the Font and Style menus intelligently.

n.b. At present, ZTextWindow is not streamable.

Class Name: ZPictWindow	Based on: ZScroller
Type:	Framework Class
Description:	Scrollable window can display Puctures and read PICT files

ZPictWindow subclasses ZScroller to read and display the contents of PICT files, or diaply Pictures from any source. Note- this is NOT a picture editor.

Class Definition

```
class ZPictWindow : public ZScroller
{
protected:
     Boolean         savePreview;
     Boolean         saveCustomIcon;
     PicHandle       thePicture;

public:

     ZPictWindow( ZCommander* aBoss, const short windID );
     ZPictWindow();
     virtual ~ZPictWindow();

     virtual void    DrawContent();
     virtual void    OpenFile( const OSType fType, Boolean isStationery = FALSE );
     virtual void    SaveFile();
     virtual void    UpdateMenus();

     virtual Boolean CanPasteType();
     virtual void    DoCopy();
     virtual void    DoPaste();

     virtual void    SetPictureFromResource( const short pictResID );

     inline     void SetSavePreview( Boolean saveIt );
     inline     void SetSaveCustomIcons( Boolean saveIt );

// streaming
     virtual void    WriteToStream( ZStream* aStream );
     virtual void    ReadFromStream( ZStream* aStream );
};
```

Data Members

<**savePreview**> is TRUE if a preview of the image is saved with the file
<**saveCustomIcon**> is TRUE if the file has a custom icon which is a miniature of the picture
<**thePicture**> is the handle to the picture data

Methods

```
     ZPictWindow( ZCommander* aBoss, const short windID );
     ZPictWindow();
     virtual ~ZPictWindow();
```

The constructors and destructor create and destroy the basic object respectively. Though not overridden by this class, you MUST call InitZWindow to initialise the window after creating it.

```
virtual void    DrawContent();
```

Overrides ZScroller to draw the picture data to the window in response to an update event.

```
virtual void    OpenFile( const OSType fType, Boolean isStationery = FALSE );
```

Reads a standard Macintosh PICT file into the internal picture handle, replacing any existing content.

```
virtual void    SaveFile();
```

Saves the current picture data to a standard PICT file, optionally adding resources for a preview and custom icons.

```
virtual void    UpdateMenus();
```

Enables the Copy command if the picture data is valid.

```
virtual Boolean CanPasteType();
```

Returns TRUE if the clipboard contains data of type ‘PICT’. This enables the Paste command.

```
virtual void    DoCopy();
virtual void    DoPaste();
```

Implements the standard Edit commands Copy and Paste. Data is saved to the clipboard as standard ‘PICT’ data.

```
virtual void    SetPictureFromResource( const short pictResID );
```

This method allows the internal image data to be set from a PICT resource rather than a file, if preferred. You would generally call this after initialising the window but before displaying it.

```
inline    void SetSavePreview( Boolean saveIt );
inline    void SetSaveCustomIcons( Boolean saveIt );
```

These methods set whether a subsequent Save or Save As will also write a preview or custom icons to the file. By default, ZPictWindow does NOT do either of these things.

```
virtual void    WriteToStream( ZStream* aStream );
virtual void    ReadFromStream( ZStream* aStream );
```

Writes and Reads the ZPictWindow to a stream.

Comments

The preview feature requires QuickTime and the Image Compression Manager. The custom icon feature is actually implemented in ZFile, and requires the “PixMapUtils.cpp” file in your project. You also need to add ZFile if you use this class.

Class Name: ZMouseTracker	Based on: ZComrade
Type:	Utility Class
Description:	Implements standard mouse dragging selection marquee

ZMouseTracker implements the dragging of a selection marquee using flicker-free techniques. It works with any ZWindow or ZScroller object. You can subclass it to implement cutsom behaviour, or listen to messages from it and implement the behaviour in the owning window.

Class Definition

```
class     ZMouseTracker   : public ZComrade
{
protected:
     ZWindow*   wOwner;
     RgnHandle  selection;
     Point      start;
     Point      current;
     Rect       constraint;
     Boolean    inScrollView;
     Boolean    autoScroll;
     short      fWidth;
     short      gridH, gridV;
     Pattern    trackPat;
     PenState   nps;

public:

     ZMouseTracker( ZWindow* anOwner, Rect* pin = NULL, Boolean willScroll = TRUE );
     virtual ~ZMouseTracker();

     virtual void    Track( const Point startPt );
     virtual void    GetSelectionBounds( Rect* aRect );
     virtual void    MakeDragRegion( RgnHandle aRgn );
     virtual void    DrawDragRegion( RgnHandle aRgn );

     virtual void    StartAction( const Point pt ) {};
     virtual void    TrackAction( const Point pt ) {};
     virtual void    CompletionAction( const Point pt ) {};

     void            SetTrackPattern( Pattern aPat );
     void            SetTrackPattern( const short patID );
     void            SetTrackPenWidth( const short aWidth );
     void            SetTrackGrid( const short h, const short v );

     void            UpdateDragRegion( RgnHandle affectedRgn );
     void            UpdateDragRect( Rect* affectedRect );

     inline     RgnHandle  GetDragRegion() { return selection; };
};
```

Data Members

<**wOwner**> is the window that “owns” this object
<**selection**> is the region of the current selection marquee

<**start**> is the point where the drag started- the anchor point of the marquee.
<**current**> is the current location of the dragged mouse.
<**constraint**> is the rectangle within which the marquee is pinned.
<**inScrollView**> is true if the owning window is a scroller rather than a plain window.
<**autoScroll**> is TRUE if the selection marquee should autoscroll the view.
<**fWidth**> is the pen width of the marque rectangle
<**gridH**> and <**gridV**> are the sizes of any grid used to constrain the drag- usually set to 1 to give a smooth selection, they can be made larger so that the selection encloses whole numbers of pixels at a time.
<**trackPat**> is the pattern used to draw the marquee
<**nps**> is the saved pen state while dragging.

Methods

```
ZMouseTracker( ZWindow* anOwner, Rect* pin = NULL, Boolean willScroll = TRUE );
```

Constructor. This creates the tracking object. Thewindow in which we are tracking must be passed as the owner. <pin> is a rectangle within which the drag will be pinned. If NULL, the bounds rect returned by the window is used, which is usually the case. <willScroll> sets whther the drag should autoscroll the window (if a scroller). Usually it should.

```
virtual void    Track( const Point startPt );
```

This method is called to handle the complete drag, starting at the point passed. This method keeps control until the mouse is released.

```
virtual void    GetSelectionBounds( Rect* aRect );
```

Returns the area enclosed by the selection- this is the rectangle defined by the original anchor point and the current mouse point.

```
virtual void    MakeDragRegion( RgnHandle aRgn );
```

Builds the drag region in <aRgn>. The default method creates a rectangular region defined by the anchor point and the current mouse point, with a frame thickness specified by <fWidth>. The region must be hollow- it is drawn by calling PaintRgn.

```
virtual void    DrawDragRegion( RgnHandle aRgn );
```

Draws the drag region by calling PaintRgn. The region is drawn in the drag pattern set, usually 50% gray.

```
virtual void    StartAction( const Point pt );
virtual void    TrackAction( const Point pt );
virtual void    CompletionAction( const Point pt );
```

These methods are called to provide status information about the drag.StartAction is called once at the beginning of the drag, TrackAction() is called repeatedly as the drag proceeds and the mouse moves to a new position, and CompletionAction(0 is called when the mouse is released. The default methods do nothing- you can override them to implement additional actions for the drag.

```
void            SetTrackPattern( Pattern aPat );
void            SetTrackPattern( const short patID );
```

Sets the pattern to draw the drag marquee in, either explicitly using a Pattern, or from the 'PAT ' resource with the ID passed. The default pattern is 50% gray.

```
void            SetTrackPenWidth( const short aWidth );
```

Sets the frame thickness of the drag marquee. This defaults to 2.

```
void            SetTrackGrid( const short h, const short v );
```

Sets the grid pitch of the drag. All points will be modified so that they "snap" to the grid of the pitch passed. The grid pitch defaults to 1.

```
void            UpdateDragRegion( RgnHandle affectedRgn );
void            UpdateDragRect( Rect* affectedRect );
```

These methods perform the same task. As the drag commences, parts of the window may be redrawn in response. This is not unusual- consider the selecting of icons in a Finder-like view. As the drag marquee intersects the icons, they will be redrawn in the selected state. Because ZMouseTracker draws by painting over the top of the view as it goes, using Xor drawing and region manipulation to eliminate flicker, when you redraw anything that intersects the drag outline, it must be immediately "patched up" or the next frame of the drag will be drawn incorrectly. These methods perform the requisite "patching up". This sounds complex but isn't really- it's the price you pay for really smooth animation though! Just keep track of the exact area you clobber when redrawing window content as you drag, and pass it to this method before handling the next frame of the drag. (Hint: you can get the region ocupied by an icon using the toolbox Icon Utilities function GetIconRegion)

```
inline    RgnHandle  GetDragRegion() { return selection; };
```

Returns the actual drag region. Note this is NOT a copy of the region, so do not dispose it!

Comments

The usual way to respond to a tracker is to receive its messages. Here they are:

```
enum
{
     msgMouseTrackStarting          = 'trk1',
     msgMouseTrackNewPosition       = 'trk2',
     msgMouseTrackComplete          = 'trk3',
     msgMouseTrackScrolledView      = 'trk4'
};
```

Class Name: MList	Based on: <none>
Type:	Mix-in Class
Description:	Implements a common object-oriented wraper to the Mac List Manager. Used by ZLMListWindow and ZListDialogItem

This class is a mix-in class that wraps the Mac List Manager. As such it provides a common API for both list windows and List box dialog item objects, and also handles many of the chores of the List Manager in a much easier way- for example, forget LDEFs, this handles all of that so you can implement any look for your list using standard overrides, etc.

Class Definition

```
class     MList
{
protected:
     ListHandle      lh;                  // list manager handle
     ZWindow*        owner;               // owning window
     Cell            lastSel;             // last selection
     Str15           searchStr;           // for finding items in a list
     long            lastKeyTime;         // time last key was typed
     short           fThresh;             // reset threshold for typing
     Boolean         usingCustomLDEF;     // true if custom LDEF installed

public:

     MList();
     MList( ZWindow* anOwner );
     virtual ~MList();

     virtual void    MLInit(    Rect* bounds,
                                Point cellSize,
                                short initialRows,
                                short initialColumns,
                                Boolean vScroll,
                                Boolean    hScroll,
                                short procID = 0 );

     virtual void    MLInit( Rect* bounds, const short listResID );

     virtual void    MLUpdate( Rect* updateRect );
     virtual void    MLClick( const Point mouse, const short modifiers );
     virtual void    MLActivate();
     virtual void    MLDeactivate();
     virtual void    MLSetSize( const short width, const short height );
     virtual void    MLScroll( const short rows, const short cols );
     virtual void    MLEnableDrawing();
     virtual void    MLDisableDrawing();
     virtual void    MLSetSelectionFlags( const char sFlags );

     virtual Boolean MLGetSelection( Cell* aCell );
     virtual void    MLSetSelection( Cell aCell );
     virtual void    MLScrollToSelection();

     virtual void    MLSetCell( Cell theCell, Ptr buf, short length );
     virtual void    MLSetCell( Cell theCell, Str255 aString );
     virtual void    MLGetCell( Cell theCell, Ptr buf, short* length );
     virtual void    MLGetCell( Cell theCell, Str255 aString );
```

```
	virtual short	MLAppendRow();
	virtual short	MLAppendCol();

	virtual short	MLAppendRowData( Ptr buf, short length );
	virtual short	MLAppendRowData( Str255 aString );
	virtual short	MLAppendColData( Ptr buf, short length );
	virtual short	MLAppendColData( Str255 aString );
	virtual short	MLAppendRowInAlphaOrder( Str255 s );

	virtual void	MLSetEmptyList();
	virtual void	MLClearCell( Cell theCell );

	virtual void	MLDeleteRow( short whichRow );
	virtual void	MLDeleteCol( short whichCol );

	virtual void	MLNewCellSelected( Cell newCell ) {};
	virtual void	MLCellDoubleClicked( Cell theCell ) {};

	virtual void	MLPreloadFromResource( const short strListID,
	                                       Boolean alphaOrder = FALSE );

	virtual void	MLGetBounds( Rect* aRect );
	virtual void	MLKeyNavigation( const char theKey, const short modifiers );

	virtual void	MLDraw1Cell( Rect* bounds, Cell theCell, Boolean hilited );
	virtual void	MLHilite1Cell( Rect* bounds, Cell theCell, Boolean hilited );
	virtual Boolean MLClickLoop() { return TRUE; };

	void		MLInstallCallbacks( Boolean customLDEF, Boolean customClikLoop );
	ListHandle	MLGetMacList() { return lh; };

};
```

Data Members

<**lh**> is the Mac ListHandle this object manages.
<**owner**> is the window that displays the list (ultimately)
<**lastSel**> is the last selection made in the list
<**searchStr**> is used to do automatic searches based on the first few characters typed
<**lastKeyTime**> is used to track whether a key extends the search or starts a new one
<**fThresh**> sets the time interval for entering search strings
<**usingCustomLDEF**> is TRUE if the standard LDEF has been replaced by our magic overridable one.

Methods

Constructor sets up the data members, destructor releases any memory.

```
	virtual void	MLInit(	Rect* bounds,
				Point cellSize,
				short initialRows,
				short initialColumns,
				Boolean vScroll,
				Boolean hScroll,
				short procID = 0 );
```

This method must be called to initialise the list after creating it. This version sets up the list explicitly. <bounds> is the bounds rectangle of the list- the visible area in the window. This includes the area occupied by the scrollbars.

<**cellSize**> is the cell dimensions in pixels, or if set to 0, will calculate a suitable cell size. <**initialRows**> is the number of rows the list has initially, and <**initialColumns**> is the number of columns. <**vScroll**> and <**hScroll**> set whether the list has vertical and horizontal scrollbars. <**procID**> is the LDEF proc to use- usually this is 0 for the standard one (we can still override it later). It's far easier to set all this up from a 'LIST' resource:

```
virtual void    MLInit( Rect* bounds, const short listResID );
```

Which this does, using the resource type 'LIST' with the id <listResID>. MacZoop comes with a ResEdit template for this resource type. The LIST resource sets up all of the above, plus one or two other small niceties.

```
virtual void    MLUpdate( Rect* updateRect );
```

Updates the list in response to an update event.

```
virtual void    MLClick( const Point mouse, const short modifiers );
```

Handles mouse clicks in the list and list's scrollbars.

```
virtual void    MLActivate();
virtual void    MLDeactivate();
```

Activates and deactivates the list.

```
virtual void    MLSetSize( const short width, const short height );
```

Sets the size of the list to the width and height passed (top, left remains where it is).

```
virtual void    MLScroll( const short rows, const short cols );
```

Scrolls the list by the number of rows and/or columns passed.

```
virtual void    MLEnableDrawing();
virtual void    MLDisableDrawing();
```

Enables and Disables drawing of thelist. This can be useful if you want to add or remove several items from the list, then refresh it on screen all in one go.

```
virtual void    MLSetSelectionFlags( const char sFlags );
```

Sets the selection behaviour for the list according to the flags. The flags are described in Inside Macintosh.

```
virtual Boolean MLGetSelection( Cell* aCell );
```

Returns the selected cell, starting at the cell initially passed. If no cell is selected, this returns FALSE and the cell is set to NO_SELECTION, otherwise TRUE is returned and <aCell> set to the selected cell. You can make repeated calls to this to check each selected cell in a multiple selection.

```
virtual void    MLSetSelection( Cell aCell );
```

Selects the passed cell, turning OFF any existing selection. This is not suitable for multiple selections.

```
virtual void    MLScrollToSelection();
```

Scrolls the list so that the selected cell is visible.

```
virtual void    MLSetCell( Cell theCell, Ptr buf, short length );
virtual void    MLSetCell( Cell theCell, Str255 aString );
```

Sets the data in the cell to that passed, either as a string or as an arbitrary buffer and length.

```
virtual void    MLGetCell( Cell theCell, Ptr buf, short* length );
virtual void    MLGetCell( Cell theCell, Str255 aString );
```

Gets the data in the cell, either as a string or arbitrary data. This is a copy- if you make changes to the data, you have to copy it back using MLSetCell.

```
virtual short   MLAppendRow();
virtual short   MLAppendCol();
```

These methods append a single row or column to the list. The new cell is empty.

```
virtual short   MLAppendRowData( Ptr buf, short length );
virtual short   MLAppendRowData( Str255 aString );
virtual short   MLAppendColData( Ptr buf, short length );
virtual short   MLAppendColData( Str255 aString );
```

These methods append a single row or column and add data to it in one go. Data may be a string or arbitrary data. These are only really suitable for one-column or one-row lists, since you can't specify which cell in the row or column if there is more than one.

```
virtual short   MLAppendRowInAlphaOrder( Str255 s );
```

This appends string data to a one-column list, inserting it into the list in alphabetical order. Provided all data is appended this way, or the list is sorted by some other means, you can take advantage of quick item selection by typing the first few letters of the item.

```
virtual void    MLSetEmptyList();
```

Removes all rows and clumns from the list. After calling this, you need to add rows and columns as required before using the list further.

```
virtual void    MLClearCell( Cell theCell );
```

Removes the contents of the cell. The cell itself is not removed.

```
virtual void    MLDeleteRow( short whichRow );
virtual void    MLDeleteCol( short whichCol );
```

Deletes a single row or column from thelist. The parameter starts at 0 for the first row or column, etc.

```
virtual void    MLNewCellSelected( Cell newCell );
```

This method is called when the selection in the list changes. You are expected to override this to do something useful.

```
virtual void    MLCellDoubleClicked( Cell theCell );
```

The cell was double-clicked. You are expected to override this to do something useful.

```
virtual void    MLPreloadFromResource( const short strListID,
                                       Boolean alphaOrder = FALSE );
```

One-column lists of strings may be preloaded froim a STR# resource. This does that. It adds as many rows as needed, reads in the strings, and can optionally sort them into alphabetical order.

```
virtual void    MLGetBounds( Rect* aRect );
```

Returns the bounds rect of the list in the window. This area includes any scrollbars the list may have.

```
virtual void    MLKeyNavigation( const char theKey, const short modifiers );
```

This method is called to navigate the list using the keyboard. The standard method responds to the arrow keys to move from cell to cell, plus command-arrow keys to move to the end of the list in the indicated direction, autoscrolling as needed all th eway. In addition, if the list contains alphabetically sorted strings, typing the first few characters of the item selects it.

```
virtual void    MLDraw1Cell( Rect* bounds, Cell theCell, Boolean hilited );
```

One method of the overridable LDEF. If you wish to override this, you can draw the cell's contents any way you like. <bounds> i sthe rectangle to do your drawing in, <theCell> is the cell to draw, and <hilited> indicates whether it's selected.

```
virtual void    MLHilite1Cell( Rect* bounds, Cell theCell, Boolean hilited );
```

Th eother method of the overridable LDEF. The cell's selection state is being changed, so redraw it as necessary.

```
virtual Boolean MLClickLoop();
```

The overridable click loop. Normally, this does nothing. You can override it to do other cunning tricks in your list, such as dragging items off using drag and drop. Returning TRUE continues the list tracking, FALSE aborts it.

```
void            MLInstallCallbacks( Boolean customLDEF, Boolean customClikLoop );
```

This method is called to switch on the overridable LDEF and clickloop features. If never called, the system LDEF or external LDEF passed to MLInit is used. By calling this early on in your list creation, a special LDEF is installed that calls back to theobject, meaning you can override the methods indicated above to implement the LDEF. The default methods mimic the system LDEF, so it's not harmful to switch this on and not override the methods.

```
ListHandle      MLGetMacList() { return lh; };
```

Returns the List Manager handle for you to do with as you will.

Class Name: ZLMListWindow		Based on: ZWindow
Type:	User Interface	
Description:	Implements a List Manager view in a window using MList	

This class implements a window containing a List Manager list, with as many rows and columns as you want. It uses the mix-in class MList to handle the List Manager, so shares a large part of its API with ZListDialogItem.

Class Definition

```
class     ZLMListWindow   : public ZWindow, public MList
{
protected:
     short          itsRows, itsCols;
     Point          cellSize;
     Boolean        inSizeOp;
     Boolean        hasVBar, hasHBar;
     Rect           bounds;
     short          headerHeight;

public:

     ZLMListWindow(ZCommander* aBoss, short windID, short rows = 0, short cols = 1 );
     virtual ~ZLMListWindow();

// window overrides
     virtual void    InitZWindow();

     virtual void    DrawContent();
     virtual void    Click( Point mouse, short modifiers );
     virtual void    Activate();
     virtual void    Deactivate();
     virtual void    SetSize( short width, short height, Boolean reDraw = TRUE );
     virtual void    Zoom( short partCode );
     virtual void    GetContentRect( Rect* contents );
     virtual void    GetIdealWindowZoomSize( Rect* zr );
     virtual void    AdjustCursor( const Point mouse, const short modifiers );
     virtual void    Type( const char theKey, const short modifiers );

// drawing the list cells

     virtual void    MLDraw1Cell( Rect* area, Cell aCell, Boolean hilited );
     virtual void    DrawHeader();
     inline  void    SetHeaderHeight( short aHeight ) { headerHeight = aHeight; };

// setting initial parameters for special lists

     virtual void    SetCellSize( short aWidth = 0, short aHeight = 0 );
     virtual void    SetLMScrollUsage( Boolean aVBar, Boolean aHBar );

     virtual void    GetBounds( Rect* aBounds) { *aBounds = bounds; };

protected:
     virtual void    DrawCellLines( Rect* area, Cell aCell );
     virtual void    CalcBounds();
};
```

Data Members

<**itsRows**>, <**itsCols**> the number of rows and columns the window has.
<**cellSize**> the size of the list cells
<**inSizeOp**> TRUE if window is processing a SetSize call.
<**hasVBar**>, <**hasHBar**> TRUE if the window has vertical and horizontal scrollbars.
<**bounds**> the bounds rect of the list.
<**headerHeight**> the height of the (optional) header area.

Methods

Most methods are straight overrides of ZWindow, and pass on calls to the relevant methods of MList. There is nothing remarkable about this so it would be tedious to enumerate them. Note that 'LIST' resources are not used with this class at present. The only methods of note are:

```
virtual void    MLDraw1Cell( Rect* area, Cell aCell, Boolean hilited );
```

Calls the MList default method to draw the cell, but in addition optionally draws dividing lines between the cells. To set this, _DRAW_LINES_BETWEEN_CELLS must be defined in the header.

```
virtual void    DrawHeader();
```

ZLMListWindows may have a header. To obtain this, set <headerHeight> to the desired height of the header. This method is then called to draw the header. If appearance is available, the default method draws the themed list view header.

```
inline  void    SetHeaderHeight( short aHeight );
```

Sets theheight of the header.

```
virtual void    SetCellSize( short aWidth = 0, short aHeight = 0 );
```

Sets the size of each cell. This should be called between object creation and InitZWindow() to be effective. Note- default cell size is based on the size of the window (horizontal width divided by number of columns) and line height of the window's font, plus a bit.

```
virtual void    SetLMScrollUsage( Boolean aVBar, Boolean aHBar );
```

Sets which scrollbars will be created. Call before InitZWindow() to be effective.

```
virtual void    DrawCellLines( Rect* area, Cell aCell );
```

Draws the cell lines if this feature is enabled.

```
virtual void    CalcBounds();
```

Calculates the bounds rectangle of the list, used amongst other things to set up the size rect and ideal zoom size for the window.

Class Name: ZGWorld		Based on: ZObject
Type:	User Class	
Description:	An object for offscreen drawing and image buffering.	

ZGWorld is an object contain a complete offscreen graphics environment. It is based on the toolbox GWorld structures but makes using them much easier. This object also handles many common chores such as conversion to and from pictures, copying with masks, search procedures, etc.

Class Definition

```
class     ZGWorld    : public ZObject
{
protected:

     GWorldPtr       gw;
     GWorldFlags     pState;
     short           lockLevel;
     Boolean         inTempMem;

public:

     // constructors

     ZGWorld( const Rect& aSize,
                const short aDepth,
                const CTabHandle aCTable = NULL,
                const Boolean makeTemp = FALSE);

     ZGWorld( const PicHandle aPicture, const Boolean makeTemp = FALSE );
     ZGWorld( const short pictID, const Boolean makeTemp = FALSE,
               const short depth = 0 );
     ZGWorld( ZGWorld* aGW, const Boolean makeTemp = FALSE );
     ZGWorld();

     // destructor

     virtual ~ZGWorld();

     // state manipulation and info

     virtual void              Lock();
     virtual void              Unlock();
     virtual void              SetPurgeable( const Boolean aPurgeState );
     virtual short             GetState();
     virtual Boolean           InMemory();

     // pixmap stuff

     virtual PixMapHandle      GetPixMap( const Boolean lock = FALSE );
     virtual void              ReallocPixMap();

     // setting the size, depth and colours

     virtual void              SetColours( const CTabHandle aCTable,
                                           const Boolean reMapColours = TRUE);
```

```
virtual void              SetDepth( const short aDepth,
                                    const CTabHandle aCTable = NULL,
                                    const Boolean dither = FALSE);
virtual void              SetSize( const Rect& aSize,
                                    const Boolean scaleImage = TRUE);

// getting info about the size, depth, colours, mem

virtual short             GetDepth();
virtual CTabHandle        GetColours();
virtual void              GetSize( Rect* aFrame );
virtual long              GetImageMemSize();
virtual short             GetColourInfo();

// copying utilities

virtual void              CopyIn( BitMap* srcBits, Rect* src, Rect* dest,
                                  short mode = srcCopy, RgnHandle mask = NULL);
virtual void              CopyOut(BitMap* destBits, Rect* src, Rect* dest,
                                  short mode = srcCopy, RgnHandle mask = NULL);
virtual void              CopyIn( ZGWorld* srcGW, Rect* src, Rect* dest,
                                  short mode = srcCopy, RgnHandle mask = NULL);
virtual void              CopyOut( ZGWorld* destGW, Rect* src, Rect* dest,
                                  short mode = srcCopy, RgnHandle mask = NULL);

virtual void              Copy( ZGWorld* aGW, short mode = srcCopy );
virtual void              Clear();

virtual void              CopyOutThroughMask( ZGWorld* destGW,
                                              ZGWorld* maskGW,
                                              Rect* src,
                                              Rect* mask,
                                              Rect* dest,
                                              short mode = srcCopy );

virtual void              CopyOutThroughMask( BitMap* destBits,
                                              ZGWorld* maskGW,
                                              Rect* src,
                                              Rect* mask,
                                              Rect* dest,
                                              short mode = srcCopy );


virtual void              CopyOutToMask( ZGWorld* destGW );

// picture utilities

virtual PicHandle         MakePicture( Rect* toFitRect = NULL,
                                       RgnHandle withMaskRgn = NULL );
virtual PicHandle         MakePicture( Rect* srcRect, Rect* destRect );
virtual void              CopyPicture(PicHandle aPicture, Rect* destRect = NULL,
                                       Boolean dither = FALSE );

// graphics port set-up

virtual void              SetPortToGW(CGrafPtr* savedPort,
                                      GDHandle* savedDevice);
virtual void              SetPortColours(RGBColor* aFore, RGBColor* aBack);
virtual void              SetPortColours();
virtual void              GetPortColours(RGBColor* aFore, RGBColor* aBack);

// icon plotting

virtual void              CopyIcon( CIconHandle anIcon, Rect* dest,
                                    short align = 0, short transform = 0);
virtual void              CopyIcon( short iconID, Rect* dest, short align = 0,
                                    short transform = 0);
```

```
      // search procedures

      virtual void              InstallSearchProc(ColorSearchUPP aSearchProc);
      virtual void              RemoveSearchProc(ColorSearchUPP aSearchProc);

      // inversion & other special effects:

      virtual void              Invert( RgnHandle maskRgn = NULL );
      virtual void              FlipHorizontal( RgnHandle maskRgn = NULL );
      virtual void              FlipVertical( RgnHandle maskRgn = NULL );

      inline GWorldPtr          GetMacGWorld() { return gw; };
      virtual void              SetMacGWorld( GWorldPtr aGW );

      // streaming

      virtual void              WriteToStream( ZStream* aStream );
      virtual void              ReadFromStream( ZStream* aStream );

protected:

      virtual void              MakeMacGWorld(  const Rect& aSize,
                                                const short aDepth,
                                                const CTabHandle aCTable,
                                                const Boolean makeTemp);
      virtual void              MakeGWorldFromPicture( const PicHandle aPic,
                                                const Boolean makeTemp,
                                                const short depth = 0 );
      virtual void              Flip( const short flipDir, RgnHandle mask );
};
```

Data Members

<**gw**> is the Mac toolbox GWorld thisobject manages.
<**pState**> is used to record the GWorld status.
<**lockLevel**> monitors multiple lock/unlock sequences
<**inTempMem**> is TRUE ifthe GWorld is created in temporary memory.

Methods

```
      ZGWorld( const Rect& aSize,
                  const short aDepth,
                  const CTabHandle aCTable = NULL,
                  const Boolean makeTemp = FALSE);

      ZGWorld( const PicHandle aPicture, const Boolean makeTemp = FALSE );
      ZGWorld( const short pictID, const Boolean makeTemp = FALSE,
                const short depth = 0 );
      ZGWorld( ZGWorld* aGW, const Boolean makeTemp = FALSE );
```

Constructors come in a variety of versions. GWorlds may be created from a set of explicit parameters, an existing Picture or PICT resource, or from another GWorld. In all cases, <makeTemp> is TRUE if you wish to create the GWorld in temporary memory. The other parameters to the constructors are: <aSize> is the bounds rect of the GWorld, <aDepth> is the pixel depth required- it must be 1, 2, 4, 8, 16 or 32. If you pass 0, the depth of the deepest monitor intersected by the bounds rect in global space is used. <aCTable> is the colour table to use. If NULL, the system colour table for the chosen depth is used. <aPicture> is an existing picture handle. The GWorld will be created to match the size, depth and colours of the picture, and the picture will be drawn in the GWorld. <pictID> is the ID of a PICT resource to load, which is used in the same way. <aGW> is another ZGworld object, which will be cloned by this one.

```
virtual ~ZGWorld();
```

Destructor frees the memory occupied by the GWorld and associated structures.

```
virtual void              Lock();
virtual void              Unlock();
```

Before a GWorld can be drawn into or drawn from, it must be locked. After performing the drawing, it should be unlocked to allow efficient memory management. These methods Lock and Unlock the GWorld. The calls can be nested- locking increments a lock counter and unlocking decrements it. The GWorld will only be actually unlocked when the count reaches zero. Thus it is important to make sure these calls are made in matched pairs. Many ZGWorld methods perform the locking for you, in particular the Copyxxx methods.

```
virtual void              SetPurgeable( const Boolean aPurgeState );
```

Sets the GWorld memory to be purgeable. Normally it is not. This allows the memory manager to discard the image from an unlocked GWorld if it needs to. It is then necessary to recreate the image by some means.

```
virtual short             GetState();
```

Returns the current state of the GWorld- see Inside Macintosh, GetPixelsState.

```
virtual Boolean           InMemory();
```

Returns FALSE if the image buffer has been purged. If you set the GWorld as purgeable, you must check this before trying to use it- you must not draw into a purged GWorld- you must recreate the image buffer first using ReallocPixMap.

```
virtual PixMapHandle      GetPixMap( const Boolean lock = FALSE );
```

Returns the pixMap of the GWorld, optionally locking it in the process.

```
virtual void              ReallocPixMap();
```

Reallocates the image buffer if it has been purged. This attempts to reallocate the memory for the image. The buffer will be filled with random data after this call- you need to clear and/or redraw the contents as necessary.

```
virtual void              SetColours( const CTabHandle aCTable,
                                      const Boolean reMapColours = TRUE);
```

This method changes thecolours of a GWorld. The GWorld should be an indexed type (2, 4 or 8 bits deep) otherwise this does nothing. The colour table passed in replaces the current one(it is copied, so the colour table passed here may be disposed by the caller).What happens to the image depends on <reMapColours>. If this is TRUE, the image will be modified so that its original colours are preserved as closely as possible using the new colours- the appearance of the image will remain fairly unchanged. If this is FALSE, the image is not remapped, and the original pixel values are unchanged. This can be used to “false colour” an image- usually the appearance of the image will change.

```
virtual void              SetDepth( const short aDepth,
                                    const CTabHandle aCTable = NULL,
                                    const Boolean dither = FALSE);
```

This method changes the pixel depth of the image, performing image conversion as needed. If you are converting to an indexed image (1, 2, 4 or 8 bits deep) you can pass a colour table for the new image. Passing NULL will use the system colour table for the depth requested. Colour tables are not used for deep images (16 or 32 bits). If converting to a lower depth, you may wish to dither the image to help avoid colour banding- set <dither> to TRUE in that case. The default is FALSE.

```
virtual void              SetSize( const Rect& aSize,
                                    const Boolean scaleImage = TRUE);
```

CHanges the size of the GWorld to the new size passed. The image may remain as it is, in which case <scaleImage> is passed as FALSE. If the GWorld is made smaller, parts of the image to the right and below the GWorld are discarded. If the new size is larger, the original image remains at the top, left of the GWorld, and the additional area is filled with white. If <scaleImage> is TRUE, the original image is scaled up or down as needed to fit the new size. n.b. if scaling down, dithering is not applied.

```
virtual short             GetDepth();
```

Returns the pixel depth of the GWorld.

```
virtual CTabHandle        GetColours();
```

Returns the images colour table, if it has one. Note that this is NOT a copy- the caller must not dispose the colour table.

```
virtual void              GetSize( Rect* aFrame );
```

Returns the current size of the GWorld.

```
virtual long              GetImageMemSize();
```

Returns the amount of memory occupied by the image buffer. If the image has been purged, this returns 0.

```
virtual short             GetColourInfo();
```

Returns the current seed value of the image’s colour table, if it has one (if not, this returns -1). This information is sometimes needed in some special applications- rarely used.

```
virtual void              CopyIn( BitMap* srcBits, Rect* src, Rect* dest,
                                  short mode = srcCopy, RgnHandle mask = NULL);
```

Copies image data from a bitMap into the GWorld. The area <src> in the bitMap is copied to the area <dest> in the GWorld, scaling as needed. The copymode is passded in <mode> and any mask region can be passed in <mask>. This is a wrapper method for CopyBits, and additionally ensures that the GWorld, etc is locked and unlocked correctly. All of the Copyxxx methods work in the same way, and are very convenient compared to CopyBits itself.

```
virtual void              CopyOut(BitMap* destBits, Rect* src, Rect* dest,
                               short mode = srcCopy, RgnHandle mask = NULL);
virtual void              CopyIn( ZGWorld* srcGW, Rect* src, Rect* dest,
                               short mode = srcCopy, RgnHandle mask = NULL);
virtual void              CopyOut( ZGWorld* destGW, Rect* src, Rect* dest,
                               short mode = srcCopy, RgnHandle mask = NULL);
```

Similarly, these methods conveniently wrap up CopyBits in various ways, moving image data from the GWorld to a bitMap, and from one GWorld to another. All take care of correctly locking and unlocking the GWorld as required.

```
virtual void              Copy( ZGWorld* aGW, short mode = srcCopy );
```

Copys the complete image in <aGW> to this one using the mode passed. If the GWorlds are different sizes, the image is scaled accordingly.

```
virtual void              Clear();
```

Erases the contents of the GWorld to white.

```
virtual void              CopyOutThroughMask( ZGWorld* destGW,
                                              ZGWorld* maskGW,
                                              Rect* src,
                                              Rect* mask,
                                              Rect* dest,
                                              short mode = srcCopy );

virtual void              CopyOutThroughMask( BitMap* destBits,
                                              ZGWorld* maskGW,
                                              Rect* src,
                                              Rect* mask,
                                              Rect* dest,
                                              short mode = srcCopy );
```

Copies the contents of the GWorld to a bitMap or another GWorld using an intermediate mask. Th emask allows the images to be blended together in an arbitrary way. The <src>, <mask> and <dfest> rects must all be the same size, but can be in different locations. The mask itself is contained in a third GWorld object, <maskGW>. This is a wrapper for CopyDeepMask, but is easier to use and manages the various locks, etc for you.

```
virtual void              CopyOutToMask( ZGWorld* destGW );
```

This method conveniently helps you make a 1-bit mask from any image. It installs a special search proc that maps all pixels to black except those that are completely white.

```
virtual PicHandle         MakePicture( Rect* toFitRect = NULL,
                                       RgnHandle withMaskRgn = NULL );
virtual PicHandle         MakePicture( Rect* srcRect, Rect* destRect );
```

These methods create a picturefrom the GWorld. The first version accepts a <toFitRect> which defines the destination size of the picture. If NULL, the picture will be the same size as the GWorld. You can also optionally pass a mask region. In the second version, the picture is sized to fir <destRect>, and the portion of the image at <srcRect> is copied and scaled to it. Both methods returned fully formed PicHandles which shouldb e disposed, etc. by the caller. These methods make light work of converting GWorlds to pictures.

```
virtual void              CopyPicture(PicHandle aPicture, Rect* destRect = NULL,
                                      Boolean dither = FALSE );
```

This is the inverse process- it copies an existing picture into the GWorld. If <destRect> is NULL, the picture is scaled to fit the entire GWorld. The picture may be optionally dithered as it is drwan to the GWorld. This is especially useful if making a small thumbnail from a larger picture. Normally, dithering a PicHandle is rather laborious, but this method handles it for you.

```
virtual void              SetPortToGW(CGrafPtr* savedPort,
                                      GDHandle* savedDevice);
```

If you want to draw into the GWorld using normal QuickDraw calls (as opposed to copying bitmaps and pictures), you must make the GWorld teh current port and graphics device. This method does that, returning thepreviously set port and device. After doing the drawing (you must also Lock and Unlock the GWorld) use the toolbox SetGWorld function to put back the original port and device.

```
virtual void              SetPortColours(RGBColor* aFore, RGBColor* aBack);
virtual void              SetPortColours();
```

Sets the fore and back colours of the GWorld's port to those passed, or if using the parameterless version, to black and white. These colours affect colourisation when performing any of the Copy operations.

```
virtual void              CopyIcon( CIconHandle anIcon, Rect* dest,
                                    short align = 0, short transform = 0);
virtual void              CopyIcon( short iconID, Rect* dest, short align = 0,
                                    short transform = 0);
```

Copies an icon into the GWorld, applying any alignment and transformation. You can copy either a CIconHandle, or pass the ID of an icon family.

```
virtual void              InstallSearchProc(ColorSearchUPP aSearchProc);
virtual void              RemoveSearchProc(ColorSearchUPP aSearchProc);
```

Attaches and removes a custom search procedure to the GWorld. These procedures are used to modify colours on the fly while copying. For example, you can easily implement brightness and contrast controls for an image using these. Refer to Inside Macintosh for moredetails.

```
virtual void              Invert( RgnHandle maskRgn = NULL );
```

Inverts all or part of the image. Note- if the image is indexed, this actually modifies the colour table, and the mask region is ignored.

```
virtual void              FlipHorizontal( RgnHandle maskRgn = NULL );
virtual void              FlipVertical( RgnHandle maskRgn = NULL );
```

Flips the image horizontally or vertically. If you pass a maskRgn, only that part of the image is flipped.

```
inline GWorldPtr          GetMacGWorld() { return gw; };
```

Returns the Mac OS toolbox GWorld that this object is managing, if you want to do your own

thing with it.

```
virtual void              SetMacGWorld( GWorldPtr aGW );
```

This method allows you to set the GWorld this object manages to one obtained from elsewhere. You must exercise extreme caution when doing this, since it simply replaces the internal <gw> data member. The old GWorld is not automatically disposed, and other status info may be in the wrong state. Very rarely used.

```
virtual void              WriteToStream( ZStream* aStream );
virtual void              ReadFromStream( ZStream* aStream );
```

Writes and reads the entire GWorld to the stream. Note that the amount of data this may add to the stream can be potentially enormous.

```
virtual void              MakeMacGWorld(  const Rect& aSize,
                                          const short aDepth,
                                          const CTabHandle aCTable,
                                          const Boolean makeTemp);
```

Called by the constructor to make the Mac GWorld using the parameters supplied.

```
virtual void              MakeGWorldFromPicture( const PicHandle aPic,
                                          const Boolean makeTemp,
                                          const short depth = 0 );
```

Called by the constructor to create the Mac GWorld by extracting the parameters from the picture and then drawing the picture to the GWorld.

```
virtual void              Flip( const short flipDir, RgnHandle mask );
```

Called by the FlipHorizontal and FlipVertical methods to do some of the work.

Comments

GWorlds can be large and very memory intensive. Many operations temporarily create entire copies of the GWorld, so memory requirements can often be double that for a single image. Because of this, be ready for any method to fail at any time. ZGWorld can throw exceptions from most methods.

You may subclass ZGWorld. One possible reason is to create your own GWorld implementation using the raw PixMap and GDevice structures. Another might be to implement a backing store type of arrangement where the GWorld uses a disk file to store the image data and allow the image to be purgeable- the image can then be recreated from disk on demand. These are just ideas- MacZoop does not provide either of these things as standard.

Class Name: ZGWorldWindow	Based on: ZPictWindow
Type:	Framework Class
Description:	Window can display a GWorld, and read and write this to a PICT file. Also handles zooming and fatbits.

ZGWorldWindow is a scroller that displays a ZGWorld image. It inherits the ability to read and write PICT files from ZPictWindow, but in addition can read GIF and JPEG files and allows the image to be zoomed and displayed as fatbits if required. This is a good basis for an image editor (but does not implement any editing by default).

Class Definition

```
class     ZGWorldWindow : public ZPictWindow
{
protected:

     ZGWorld*   itsOffscreen;        // the GWorld
     short      scale;               // display zooming scale
     Boolean    fatBitsGrid;         // TRUE if fatbits grid overlaid

public:

     ZGWorldWindow( ZCommander* aBoss, const short windID );
     ZGWorldWindow();
     ~ZGWorldWindow();

     virtual void    DrawContent();
     virtual void    OpenFile( const OSType fType, Boolean isStationery = FALSE );
     virtual void    SaveFile();

     virtual void    HandleCommand( const long theCommand );
     virtual void    UpdateMenus();

     virtual Boolean AcceptsFlavour( const OSType aFlavour );
     virtual void    Drop( const OSType flavour, const Ptr data,
                           const long dataSize, const DragReference theDrag = NULL );

     virtual void    DoCopy();
     virtual void    DoPaste();

     inline    ZGWorld*   GetGWorld() { return itsOffscreen; };
     virtual   void       SetGWorld( ZGWorld* aGW );

     virtual void    MakePaletteForWindow();

     virtual void    SetScale( const short aScale = 100 );
     inline short    GetScale() { return scale; };

     virtual void    ZoomToPoint( const Point clickPt, Boolean isZoomOut = FALSE );
     virtual void    ZoomToCentre( Boolean isZoomOut = FALSE );

     virtual void    GetName( Str255 name );
     virtual void    SetTitle( Str255 name );

     virtual void    SetPictureFromResource( const short pictResID );
```

```
// new 2.0.1 convenience interface, computes bounds, etc:

     virtual void    InstallImage( PicHandle aPic );
     virtual void    InstallImage( const short picResourceID );
     virtual void    InstallImage( ZGWorld* aGWorld );

// streaming:

     virtual void    WriteToStream( ZStream* aStream );
     virtual void    ReadFromStream( ZStream* aStream );

protected:

     virtual void    MakeGWorld();
     virtual void    CalcSourceRect( Rect* aSrcRect );
     virtual void    CalcBoundsRect( Rect* aBoundsRect );
     virtual void    CalcFatBitsGridMask( const Rect& destRect, RgnHandle maskRgn );
};
```

Data Members

<**itsOffscreen**> the ZGWorld object this window displays
<**scale**> the zoom scale of the window, in percent
<**fatBitsGrid**> TRUE if the enlarged image is overlaid with a fatbits grid.

Methods

```
     ZGWorldWindow( ZCommander* aBoss, const short windID );
     ZGWorldWindow();
     ~ZGWorldWindow();
```

Constructors and destructor perform the usual. The contrsuctor parameters are as for any window. Note that this window type always has both horizontal and vertical scrollbars.

```
     virtual void    DrawContent();
```

Draws the window contents by copying the ZGWorld image to the window, taking into account the scaling factor and scroll offset, etc.

```
     virtual void    OpenFile( const OSType fType, Boolean isStationery = FALSE );
```

This method opens PICT, GIF and JPEG files into the image.

```
     virtual void    SaveFile();
```

Saves the GWorld image to a PICT file (n.b. cannot write GIF and JPEG files at the time of writing).

```
     virtual void    HandleCommand( const long theCommand );
```

Handles the commands for this window, which are zoom in, zoom out and fat bits toggle.

```
     virtual void    UpdateMenus();
```

Enables the zoom in, zoom out and fatbits commands.

```
virtual Boolean AcceptsFlavour( const OSType aFlavour );
```

Accepts drag flavours for PICT data and files (flavour ‘hfs ’). This allows image files to be dragged and dropped in the window to display them, and PICT clipping to be dragged in also.

```
virtual void    Drop( const OSType flavour, const Ptr data,
                      const long dataSize, const DragReference theDrag = NULL );
```

Handles the drop of image files and PICT clippings in the window.

```
virtual void    DoCopy();
virtual void    DoPaste();
```

Copies and pastes PICT data from the image to the clipboard. The copy command places a copy of the image as PICT data on the clipboard, Pasting replaces the image with any PICT data on the clipboard.

```
inline    ZGWorld*   GetGWorld();
```

Returns the ZGWorld object the window is using.

```
virtual    void       SetGWorld( ZGWorld* aGW );
```

Allows the window to take on the display of an already existing ZGWorld object. The old one is not deleted however, and this does not recompute the bounds or other parameters. Generally, you should use the InstallImage() method instead.

```
virtual void    MakePaletteForWindow();
```

Sets up the palette for the window to best display the image on bit-depth challenged monitors. If the image is indexed, its color table is used to make the palette, otherwise the Picture Utilities are called to make a palette using the most popular colours. Note that if your application has floating windows, the palette will not be activated correctly- you will need to deal with this somehow, perhaps by attaching the palette to a dummy window that is kept in front at all times, but hidden behind the menu bar (a common approach to the problem).

```
virtual void    SetScale( const short aScale = 100 );
```

Sets the zoom scale of the window. The <scale> parameter represents the zoom scale in percent. This can range between 25% and 3,200%. Scrollbars are adjusted to maintain sensible scrolling rates at larger scales.

```
inline short    GetScale();
```

Returns the current scale.

```
virtual void    ZoomToPoint( const Point clickPt, Boolean isZoomOut = FALSE );
```

An easy method to support “magnifying glass” type tools. This zooms the image in or out by a factor of two, centring the image on the point passed. You can call this from your ClickContent method to implement a zooming tool.

```
virtual void    ZoomToCentre( Boolean isZoomOut = FALSE );
```

As above, but zooms the view maintaining the current centre point. Used to implement the zoom command.The image is zoomed by a factor of two by this method.

```
virtual void    GetName( Str255 name );
```

Returns the title of the window. In fact, this is the filename of the file associated, since the title displayed in the title bar is modified to also show the current zoom scale.

```
virtual void    SetTitle( Str255 name );
```

Sets the title of the window to that passed, but also appends the zoom scale. The title displayed is of the form “MyImage @200%”, etc.

```
virtual void    SetPictureFromResource( const short pictResID );
```

Overrides the ZPictWindow mehod to set up the GWorld from the PICT resource requested. It’s usually more convenient to use the InstallImage() method.

```
virtual void    InstallImage( PicHandle aPic );
virtual void    InstallImage( const short picResourceID );
virtual void    InstallImage( ZGWorld* aGWorld );
```

Installs an image from a variety of sources, correctly recalculating the scrollbounds, etc. The image may come from a PicHandle, a PICT resource, or an existing ZGWorld object. Note that the ZGworld version of this does NOT discard the old ZGWorld object. Usually, you use one of these methods after creating the window to set up which image it displays, so this is not an issue. However, if you call it later on, you may need to take steps to perform a little bit of memory management.

```
virtual void    WriteToStream( ZStream* aStream );
virtual void    ReadFromStream( ZStream* aStream );
```

Writes and reads the window and image to the stream.

```
virtual void    MakeGWorld();
```

Creates the internal ZGWorld from the attached picture (creates the GWorld using the <thePicture> data member which it inherits from ZPictWindow).

```
virtual void    CalcSourceRect( Rect* aSrcRect );
```

Computes the source rectangle in the GWorld corresponding to the content rect of the window, taking into account the scale factor.

```
virtual void    CalcBoundsRect( Rect* aBoundsRect );
```

Computes the bounds rect of the scroller based on the size of the ZGWorld, taking into account the scaling factor.

```
virtual void    CalcFatBitsGridMask( const Rect& destRect, RgnHandle maskRgn );
```

Creates the fatBits mask region. This is drawn automatically if the fatbits flag is on and the image is zoomed up to at least 401%. Note that the fatBits region can become complex if there

are a lot of pixels displayed, and in this case may not appear. This is a limitation of regions.

Comments

This is intended to be a fairly high-level, self-contained object. It provides an excellent basisi for an image editor- you only need to add the editing tools!

The features of the window are available via the commands it defines, which are assigned the following values:

```
enum
{
	kCmdFlipHorizontal = 188,
	kCmdFlipVertical,
	kCmdInvert,
	kCmdZoomIn,
	kCmdZoomOut,
	kCmdShowHideGrid
};
```

Note that as well as zooming, it also supports image inversion and flipping, courtesy of ZGWorld.

This class requires ZFile, ZGIFFile and ZJPEGFile. The image commands are also undoable, and for this ZUndoIPTask is required. This particular undo task is not documented in the class reference, but its operation is clear enough- examine the source code if you need to understand it.

Class Name: ZFile	Based on: ZComrade
Type:	Framework Class
Description:	General file handling class, basis for all other file classes.

ZFile implements the basic operations of all files. It can be used as is, or subclassed for specialist operations. Other MacZoop file classes are based on this. It also implements easy safe-saving.

Class Definition

```
class     ZFile : public ZComrade
{
protected:

     short     refNum;         // data fork ref num when open
     short     resRefNum;      // resource fork ref num when open
     Boolean   isSafeSave;     // TRUE if this was a safe-save Write
     OSType    itsType;        // file type
     OSType    itsCreator;     // file creator- initialised to app creator type
     FSSpec    itsSpec;        // file spec (name and location)
     FSSpec    ssFSpec;        // temp file spec for safe-save
     long      tempFID;        // temp folder ID
     short     tempFVolID;     // temp folder volume

public:

     ZFile( const FSSpec& aSpec );
     ZFile( Str255 fName );
     ZFile();
     ~ZFile();

// file opening and closing
     virtual void    Open( SInt8 permission = fsCurPerm );
     virtual void    Close();
     virtual void    OpenSafe();

// creating/destroying a new file on disk
     virtual void    Create();
     virtual void    Discard();

// accessing the resource fork
     virtual void    OpenResFork( SInt8 permission = fsCurPerm );
     virtual void    CloseResFork();
     virtual void    CreateResFork();
     virtual void    SetResFork( short* curRes );

// reading and writing file data
     virtual void    Read( Ptr inBuffer, long* howMuch );
     virtual void    Write( Ptr outBuffer, long* howMuch );
     virtual void    Read( Handle aHandle );
     virtual void    Write( Handle aHandle );

// file positioning and info
     virtual long    GetMark();
     virtual void    SetMark( const long aMark );

     virtual OSType  GetType();
     virtual void    SetType( const OSType aType );
```

```
      virtual void    SetCreator( const OSType aCreator );

      virtual long    GetLength();
      virtual void    SetLength( const long aLength );

      virtual void    GetFSSpec( FSSpec* aSpec );
      virtual void    GetInfo( FInfo* fi );
      virtual void    SetInfo( FInfo* fi );

// fork info
      virtual Boolean HasResFork();
      virtual Boolean HasDataFork();

// disk file info
      virtual Boolean IsReal();
      virtual Boolean IsLocked();
      virtual Boolean IsOpen();

      inline    short GetRefNumber() { return refNum; };
      inline    short GetResourceRefNumber() { return resRefNum; };

// making custom icons for the file:

      virtual void    MakeCustomIcon( PicHandle srcImage );
      virtual void    MakeCustomIcon( ZGWorld* srcImage );

      virtual void    GetDebugInfoString( Str255 s );
protected:

      virtual void    InitFile();
      virtual Handle  ConstructCustomIconSuite( PicHandle srcPic );
      virtual void    SaveCustomIconSuite( Handle icnSuite );

      void            GetTempFolderID();
};
```

Data Members

<**refNum**> is the reference number of the open file, or is set to NOT_OPEN if the file is closed.
<**resRefNum**> is the reference number of the resource fork of the file, if open
<**isSafeSave**> is TRUE if the file is currently in safe-save mode
<**itsType**> is the file's type
<**itsCreator**> isthe file's creator
<**itsSpec**> is the file spec of the file on disk
<**ssFSpec**> is the file spec of the temporary file used for safe-save
<**tempFID**> i sthe ID number of the "temporary items" folder
<**tempFVolID**> is the volume ID of the "temporary items" folder

Methods

```
      ZFile( const FSSpec& aSpec );
      ZFile( Str255 fName );
      ZFile();
```

Constructors set up the file object. You can pass a complete file spec, or just the file name. In this case, the file is assumed to be in the default folder- which will be where the application is unless something changed it. The default constructor sets up a null filespec.

```
      ~ZFile();
```

Destructor closes all open paths to the file before the object is deleted.

```
virtual void    Open( SInt8 permission = fsCurPerm );
```

Opens the file ready for reading or writing, according to the passed permission. The default is fsCurPerm. If the file does not exist, this will throw an error.

```
virtual void    Close();
```

Closes the file's data fork. In addition, if a safe-save was being performed, this swaps the tempporary file for the original one and discards the old file.

```
virtual void    OpenSafe();
```

Opens the file for writing in safe-save mode. This opens a temporary filein the "temporary items" folder. Subsequent writes will write data to this file. When it is closed, the file swap will be performed. The filename of the temp file is generated algorithmically- if a failure occurs and the file cannot be written, the Mac OS will clean up the file on the next start-up, moving it to the wastebasket.

```
virtual void    Create();
```

Creates the file on disk. This must be done for a non-existent file before it is opened.

```
virtual void    Discard();
```

Deletes the file from the disk.

```
virtual void    OpenResFork( SInt8 permission = fsCurPerm );
```

Opens the resource fork for reading and writing using the passed permission. You should use Resource Manager functions or ZResourceFile to access data in the resource fork.

```
virtual void    CloseResFork();
```

Closese the resource fork of the file.

```
virtual void    CreateResFork();
```

Creates a resource fork for the file.

```
virtual void    SetResFork( short* curRes );
```

Sets the resource fork of the file as the current resource file, returning the previous one. After accessing data in the resource fork using the Resource Manager (or ZResourceFile), you should restore the original resource file using UseResFile().

```
virtual void    Read( Ptr inBuffer, long* howMuch );
virtual void    Read( Handle aHandle );
```

Reads data from the file to an arbitrary buffer or a Handle. <inBuffer> is a pointer to the storage area for the data, and <howMuch> is a pointer to a long, indicating how many bytes to read. The actual number of bytes read is returned in <howMuch>. You can also pass a Handle, which will

be sized to accommodate the entire file and the data read into it. You should avoid using this technique with large files.

```
virtual void    Write( Ptr outBuffer, long* howMuch );
virtual void    Write( Handle aHandle );
```

Writes the data in <outBuffer> or <aHandle> to the file. The data will be written starting at the current mark, and the file will be extended if needed to accommodate the data.

```
virtual long    GetMark();
```

Returns the current position where data will be read or written.

```
virtual void    SetMark( const long aMark );
```

Sets the mark to the position passed. This starts at 0 for the first byte in the file, and has a maximum of the file length - 1.

```
virtual OSType  GetType();
virtual void    SetType( const OSType aType );
```

Gets and Sets the file type of the file. If the file exists on disk, it will be updated to the new type. If not, it will have this type when it is created.

```
virtual void    SetCreator( const OSType aCreator );
```

Sets the creator code of the file. Normally, this is initialised to <gAppSignature>, but if you wish to create files to open in other applications, you can use this method to set the creator code. If the file exists on disk, it is updated to the new creator.

```
virtual long    GetLength();
```

Returns the length of the file in bytes.

```
virtual void    SetLength( const long aLength );
```

Sets the length of the file to the length passed. After writing data to a file, you should set the length of the file to the current mark when you're done, but before closing the file. This is done if you Write a complete handle to the file, but if you are writing stuff piece by piece, you need to do this.

```
virtual void    GetFSSpec( FSSpec* aSpec );
```

Returns the filespec of the file.

```
virtual void    GetInfo( FInfo* fi );
```

Returns the Finder info for the file. If the file does not exist on disk, this will throw an error.

```
virtual void    SetInfo( FInfo* fi );
```

Sets the Finder Info for the file. The file must exist on disk.

```
virtual Boolean HasResFork();
```

Returns TRUE if the file has a resource fork. If the file does not exist or has no resource fork, it returns FALSE

```
virtual Boolean HasDataFork();
```

Similarly for the data fork.

```
virtual Boolean IsReal();
```

Returns TRUE if the file really exists on the disk, otherwise FALSE.

```
virtual Boolean IsLocked();
```

Returns TRUE if the file is locked. This is the software lock for the file- it does not indicate if the file is on a locked or read-only volume.

```
virtual Boolean IsOpen();
```

Returns TRUE if the file is open.

```
inline     short GetRefNumber();
```

Returns the file reference number for the data fork.

```
inline     short GetResourceRefNumber();
```

Returns the file reference number for the resource fork.

```
virtual void    MakeCustomIcon( PicHandle srcImage );
virtual void    MakeCustomIcon( ZGWorld* srcImage );
```

Creates custom icons for the file based on the picture of ZGWorld image passed. Complete suites are created using the PixMapUtils.cpp source code. The custom icons are automatically added to the file's resources and the "custom icon" bit is set so they will be displayed by the Finder.

```
virtual void    GetDebugInfoString( Str255 s );
```

Returns readable information that can be used by runtime debuggers and inspectors, etc.

```
virtual void    InitFile();
```

Common initialisation called by all constructors.

```
virtual Handle  ConstructCustomIconSuite( PicHandle srcPic );
```

Constructs the icon suite from the picture.

```
virtual void    SaveCustomIconSuite( Handle icnSuite );
```

Saves the icon suite to the file.

```
void            GetTempFolderID();
```

Obtains the volume and directoy ID of the temporary items folder used for safe-save.

Comments

ZFile is designed to be used as a stack or heap object- it may be very conveniently used as a stack object.

Files must be created and opened before they can be read and written. The usual set up is (stack example):

```
ZFile     f( myFileSpec );

if ( ! f.IsReal())
     f.Create();

f.Open();
f.Write(...);
```

Using a stack object means the file is automatically closed when it goes out of scope.

Safe saving is easy. Simply call OpenSafe() instead of Open() and write as normal. The Close method will perform the swap.

Class Name: ZFolderScanner	Based on: ZFile
Type:	Framework Class
Description:	Allows a folder or tree of folders to be scanned, and each file processed.

ZFolderScanner is a convenient way to iterate through a folder, a tree of folders or an entire volume, and examine each file encountered. You can subclass or listen to this object to actually process the files as they are returned. This object can optionally use a progress dialog to inform the user what it is doing.

Class Definition

```
class     ZFolderScanner : public ZFile
{
protected:
     short          curDepth;
     short          searchDepth;
     short          sProg;
     long           topDirID;
     Boolean        useProgressDialog;
     ZProgress*     itsPD;
     CInfoPBRec     pb;
     Str31          fName;

public:
     ZFolderScanner( const FSSpec& rootFolder );
     ZFolderScanner();

     ~ZFolderScanner();

     virtual void    SetSearchDepth( const short aSearchDepth );
     virtual Boolean PickFolder();
     virtual void    ScanFolder();

     inline    void SetUseProgress( Boolean useIt ) { useProgressDialog = useIt; };

protected:
     virtual void    Scan1Folder( const long dirID );
     virtual void    Process1File( const FSSpec& aSpec, const OSType fType );
     virtual void    Process1Folder( const FSSpec& aSpec );
};
```

Data Members

<**curDepth**> is the depth in the folder tree currently reached
<**searchDepth**> is the maximum depth to go down to in the folder tree
<**sProg**> is not used
<**topDirID**> is the directory ID of the start of the search (top of folder tree)
<**useProgressDialog**> is TRUE if a progress dialog is displayed while scanning
<**itsPD**> is the progress dialog, if used
<**pb**> is the parameter block used to track the search
<**fName**> is the current file name buffer

Methods

```
ZFolderScanner( const FSSpec& rootFolder );
ZFolderScanner();
```

Constructors set up the object. The fileSpec passed should be for a folder. It is more common to use the parameterless version and set the folder later. If you don't later set the folder, this defaults to the entire start-up disk.

```
virtual void    SetSearchDepth( const short aSearchDepth );
```

Sets the search depth for the scan. You can pass 0 to scan a single folder and ignore any folders with in (the default), or any value to scan to that depth (e.g. passing 1 searches the folder and any within, but not any further than that). If you pass -1, the search goes as deep as necessary to search all folders in the tree.

```
virtual Boolean PickFolder();
```

Displays a user-interface for selecting a folder to scan. This uses Navigation Services if available, or uses a Custom Standard File dialog otherwise. The picked folder is set in the <itsFilespec> data member, and it returns TRUE i fthe user chose a folder, FALSE if they cancelled.

```
virtual void    ScanFolder();
```

Actually performs the scan. Keeps control until the scan is completed.

```
inline    void SetUseProgress( Boolean useIt );
```

Sets whether a progress dialog is displayed during the scan or not. The default is to use one, sincethe scan can be lengthy, and it's a good idea to cooperate with other processes during a scan.

```
virtual void    Scan1Folder( const long dirID );
```

Scans a single folder with the directory ID passed. You should not directly use this method- call ScanFolder() to initiate a scan, and override Process1File or Process1Folder to deal with the files and folders encountered.

```
virtual void    Process1File( const FSSpec& aSpec, const OSType fType );
```

Each file is passed to this method. The default method update sthe progress dialog with the file's name and broadcasts a message. You can override it, or listen for the message.

```
virtual void    Process1Folder( const FSSpec& aSpec );
```

This is called for each folder encountered, before it is itself scanned. Note that there's a bug or feature in the design- this is only called for folders that will be scanned anyway, so if the search depth is 0, you won't get to know about any folders in the top folder in this method. This may change in future, but that's how it is right now.

Comments

ZFolderScanner is easy to use- here's a typical code example which will scan the entire start-up disk or any other folder as picked by the user.

```
void ScanDisk()
{
     ZFolderScanner  fs;

     fs.SetSearchDepth( kScanEveryFolderInHierarchy );

     if ( fs.PickFolder())
          fs.ScanFolder();
}
```

The progress dialog displayed uses the "indeterminate" style, since the number of files is not known in advance. By default, the progress message is set to each file encountered.

ZFolderScanner messages

You can listen to this object to process the files rather than subclass it if you prefer. Here's what to listen for:

```
enum
{
     msgFolderscanProcess1File       = 'fscf',
     msgFolderscanProcess1Folder     = 'fscd'
};
```

In both cases, the message data is a pointer to the filespec of the file or folder.

This class requires ZProgress and FileMgrUtils.cpp in your project.

Class Name: ZResourceFile	Based on: ZFile
Type:	Framework Class
Description:	A convenient way to read and write resources to a file.

ZResourceFile adds Resource Manager handling to the basic resource fork methods of ZFile. It is slightly safer to use than the raw resource manager, since it is careful about preserving the current resource file. Thus you can have several resource files open, and generally not worry about managing the resfile chain.

Class Definition

```
class     ZResourceFile : public ZFile
{
public:

     ZResourceFile( const FSSpec& aSpec );
     ZResourceFile( Str255 fName );
     ZResourceFile( const short resRefNum );

// getting resources

     virtual Handle       ReadResource( const ResType aType, const short resID,
                                        const Boolean detachIt = FALSE );
     virtual Handle       ReadResource( const short index, const ResType aType,
                                        const Boolean detachIt = FALSE );
     virtual Handle       ReadResource( const ResType aType, Str255 resName,
                                        const Boolean detachIt = FALSE );

// resource info

     virtual Boolean      OwnsResource( Handle aResHandle );
     virtual Boolean      HasResource( const ResType aType, const short resID );
     virtual Boolean      HasResType( const ResType aType );

     virtual short        TotalResources();
     virtual short        CountResources( const ResType aType );

     virtual short        GetRFAttributes();

     virtual void         GetResourceInfo( Handle aResHandle, ResType* itsType,
                                           short* itsID, Str255 itsName );
     virtual void         GetResourceInfo( Handle aResHandle, ResType* itsType,
                                           short* itsID );

// writing resources

     virtual void         WriteResource( Handle aResHandle, const ResType aType,
                                           const short resID );
     virtual void         WriteResource( Ptr data, const long length,
                                        const ResType aType, const short resID );

// deleting resources

     virtual void         DeleteResource( Handle aResHandle );
     virtual void         DeleteResource( const ResType aType, const short resID );
     virtual void         DeleteAll();
```

```
// other stuff

     virtual void         ResourceModified( Handle aResHandle );
     virtual void         Flush();

};
```

Data Members

none.

Methods

```
     ZResourceFile( const FSSpec& aSpec );
     ZResourceFile( Str255 fName );
     ZResourceFile( const short resRefNum );
```

Constructors as for ZFile, except you can also pass the reference number of an existing, open resource file and have this object manage it.However, you should take care usually not to close files set up liek this. Also, be aware the fileSpec will be incorrect.

```
     virtual Handle       ReadResource( const ResType aType, const short resID,
                                        const Boolean detachIt = FALSE );
     virtual Handle       ReadResource( const short index, const ResType aType,
                                        const Boolean detachIt = FALSE );
     virtual Handle       ReadResource( const ResType aType, Str255 resName,
                                        const Boolean detachIt = FALSE );
```

Read a resource, optionally detaching it. Returns the handle to the resource. You can obtain resources by type and ID, index and type, or type and name. If th eresource is detached, the caller is responsible for disposingof it. If not, the caller should call ReleaseResource on it when done with it.

```
     virtual Boolean      OwnsResource( Handle aResHandle );
```

Returns TRUE if the resource came from this file. Note that if the resource was detached, this will return FALSE

```
     virtual Boolean      HasResource( const ResType aType, const short resID );
```

Returns TRUE if this file contains a resource with the type and ID passed.

```
     virtual Boolean      HasResType( const ResType aType );
```

Returns TRUE if the file has any resources of the given type.

```
     virtual short        TotalResources();
```

Returns the number of different types of resources in the file.

```
     virtual short        CountResources( const ResType aType );
```

Returns the number of resources of the given type in the file.

```
     virtual short        GetRFAttributes();
```

Returns the attribute flags of the resource fork. Very rarely required.

```
virtual void        GetResourceInfo( Handle aResHandle, ResType* itsType,
                                     short* itsID, Str255 itsName );
virtual void        GetResourceInfo( Handle aResHandle, ResType* itsType,
                                     short* itsID );
```

Gets information about a resource. Given a resource handle, this returns its type and ID number, and also its name if using the first version of the method.

```
virtual void        WriteResource( Handle aResHandle, const ResType aType,
                                   const short resID );
virtual void        WriteResource( Ptr data, const long length,
                                  const ResType aType, const short resID );
```

Writes data to the file as a resource. This does various things depending on what the data is. If the resource already exists in the file, it is merely marked as modified. If new data is supplied using the second variant of the method, the existing resource's data is overwritten with the new data. If the resource does not exist in either case, it will be created and added to the file. You can call this method pretty much as you wish and it will do the right thing. Any changes to existing resources should be flushed to the disk using Flush().

```
virtual void        DeleteResource( Handle aResHandle );
virtual void        DeleteResource( const ResType aType, const short resID );
```

Deletes the resource from the file. The handle is disposed of.

```
virtual void        DeleteAll();
```

Removes and discards all of the resources from the file, setting the resource fork length to zero. WARNING: do not use if you are using this to manage an existing open file. It requires a valid FSSpec.

```
virtual void        ResourceModified( Handle aResHandle );
```

Marks the resource as modified. A later flush will writ ethe changed data to the file.

```
virtual void        Flush();
```

Writes out all changed data to the file.

Class Name: ZPrefsFile	Based on: ZResourceFile
Type:	Framework Class
Description:	A simple way to manage your application's prefs file

ZPrefsFile is a file (both data and resource) which knows how to locate itself in the Preferences folder. In addition, it can also work out its own filename based on the application's name. Once created and opened, you can use it in any way you wish to implement prefs for your application.

Class Definition

```
class     ZPrefsFile : public ZResourceFile
{
public:
     ZPrefsFile( Str255 fName );
     ZPrefsFile();
};
```

Data Members

none.

Methods

```
     ZPrefsFile( Str255 fName );
     ZPrefsFile();
```

Constructors. You can either pass an explicit name to be used for the file, or nothing, in which case the application's name is used, appended with " prefs". In either case, the file is located in the current Prefs folder on the startup disk. The file type is set to 'pref' and the creator is set to your application's creator.

Comments

It is usual to assign the global <gPrefsFile> to a file of this kind, usually early on in your application's start-up (in fact ZApplication provides a method, ReadPrefs, for doing this). After creating the prefs file object, you should create and open it. This is not done automatically since you are free to use either the resource fork or the data fork or both. Other parts of MacZoop can writ eprefs for you- for example, ZWindowManager can write Wpos and Dpos resources to save a window or dialog's screen location in the prefs.

Class Name: ZGIFFile		Based on: ZFile
Type:	Framework Class	
Description:	A file that can read GIF files into a GWorld	

ZGIFFile can parse GIF files and produce the image in a ZGWorld. It is read-only.

Class Definition

```
class     ZGIFFile : public ZFile
{
public:

     ZGIFFile( const FSSpec& aSpec );

     virtual void    Read( ZGWorld*   aGWorld );
     virtual void    Read( GWorldPtr* aGWorld );
};
```

Data Members

none.

Methods

Constructor as for ZFile.

```
     virtual void    Read( ZGWorld*   aGWorld );
     virtual void    Read( GWorldPtr* aGWorld );
```

Reads the GIF file into a ZGWorld object or a native Mac GWorld structure. The GWorld is sized to the file's image and set up to the right depth and colours. The file's data fork must be opened before calling this method.

Class Name: ZJPEGFile	Based on: ZFile
Type:	Framework Class
Description:	A file that can read JPEG/JFIF files into a GWorld

ZJPEGFile can parse JPEG and JFIF files into a GWorld. It is read-only.

Class Definition

```
class     ZJPEGFile : public ZFile
{
public:

     ZJPEGFile( const FSSpec& aSpec );

     virtual void    Read( ZGWorld*   aGWorld );
     virtual void    Read( GWorldPtr* aGWorld );
};
```

Data Members

none.

Methods

Constructor as for ZFile.

```
     virtual void    Read( ZGWorld*   aGWorld );
     virtual void    Read( GWorldPtr* aGWorld );
```

Reads the JPEG file into a ZGWorld object or a native Mac GWorld structure. The GWorld is sized to the file's image and set up to the right depth and colours. The file's data fork must be opened before calling this method.

Class Name: ZUndoTask	Based on: ZObject
Type:	Framework Class
Description:	Generic base class for undoable tasks.

ZUndoTask provides a framework for undoable tasks. It is handled by both ZWindow and ZApplication. Normally you will subclass this to implement the actual udo processing- this provides a common mechanism.

Class Definition

```
class     ZUndoTask  : public ZObject
{
protected:
     Str63     taskString;     // name of the task- menu shows "Undo <taskString>"
     ZWindow*  itsTarget;      // window the task is intended for
     Boolean   undone;         // if undone (if TRUE, menu shows Redo)
     Boolean   isFirstTask;    // TRUE if this is the first task since file saved

public:
     ZUndoTask( Str63 aTaskString, ZWindow*    aTarget );
     ZUndoTask( const short strListID, const short strListIndex, ZWindow* aTarget );
     virtual ~ZUndoTask() {};

     virtual void    Do();
     virtual void    Undo();
     virtual void    Redo();
     virtual void    GetTaskString( Str63 aTaskStr );

     inline Boolean  IsUndone() { return undone; };
     inline ZWindow* GetUndoTarget() { return itsTarget; };

     inline void     SetIsFirstTask() { isFirstTask = TRUE; };
};
```

Data Members

<**taskString**> is the user-readable name of the task as it appears in the Edit menu.
<**itsTarget**> is the window where the task originated and will be applied
<**undone**> is TRUE if the task has been undone once- next call will be Redo, etc.
<**isFirstTask**> is TRUE if this is the first task the window has seen sincethe file was last saved. This allows the “Save Changes?” alert to be suppressed if the first task is then undone.

Methods

```
     ZUndoTask( Str63 aTaskString, ZWindow*    aTarget );
     ZUndoTask( const short strListID, const short strListIndex, ZWindow* aTarget );
```

Constructors create the task object and assign its task string and target. The string can be passed explicitly, or else looked up from a STR# resource with the ID and index passed.

```
     virtual void    Do();
```

Do is called to execute the task initially. This sets the target's dirty flag. It's up to you whether the undo task performs the initial operation- sometimes this makes sense, other times it's easier to just perform the operation directly and not call the task's Do method. The Do() method is not called automatically except by the default Redo() method.

```
virtual void    Undo();
```

Called when the user chooses the Undo <task> menu command in the Edit menu. The task should do whatever is necessary to put back the state of the window to what it was when the task was created. If this is the window's first task, the dirty flag is cleared.

```
virtual void    Redo();
```

The default method simply calls the Do() method.

```
virtual void    GetTaskString( Str63 aTaskStr );
```

Returns the task's string. Called by ZApplication to set up the Undo menu command wording.

```
inline Boolean  IsUndone();
```

Returns TRUE if the task is in the undone state. If so, ZApplication sets the menu command to "Redo <taskname>"

```
inline ZWindow* GetUndoTarget();
```

Returns the target for the task. Note that if the target is not frontmost, the undo command is greyed out.

```
inline void     SetIsFirstTask();
```

Set by the window's SetTask method. Do not use directly.

Comments

Undo tasks are passed to the application via ZWindow's SetTask method (see ZWindow). Undo tasks are automatically discarded when the target window is closed or saved to disk.

Implementing Undo need not be hard, but requires some thought. This object, in conjunction with ZApplication, only provides the basic user interface for undo. The hard work is left to you! To implement Undo, you need to store enough information about the state of your data model to be able to put it back how it was when the task was created. One way to do this is to store a complete copy of the data model in the original state in the undo task, then swap the old and new copies to perform the undo. This may be inefficient if your data model is large. If you want to provide Undo in your application, it should be designed in from the beginning.

Multiple Undos can be implemented in a subclass of ZApplication. This can then maintain a pair of stacks of ZUndoTask objects- each time undo is called, the object is moved from the undo stack to the redo stack. New tasks are pushed onto the undo stack and the redo stack is discarded and reset.

ZUndoIPTask shows you one way to implement Undo (see ZGWorldWindow).

Class Name: ZProgress	Based on: ZDialog
Type:	Framework Class
Description:	Provides standard user interface for progress reporting.

ZProgress provides a standard dialog for indicating the progress of a lengthy procedure. As well as making this easy, it also handles the cancel/stop button and cooperates with oher processes as it does so (while this may make your lengthy procedure take slightly longer than it otherwise would have done, it also means your application is a good Mac citizen!).

Class Definition

```
class     ZProgress : public ZDialog
{
protected:
     short          theType;
     long           createTime;
     long           deferTime;
     long           estTicksToFinish;
     Boolean        fAbort;
     short          evtProcRatio;
     short          evtProcCount;

public:

     ZProgress( ZCommander* aBoss,
                    const short dialogID = kStdProgressResID,
                    const long maxValue = 100,
                    const short pType = kCancelType,
                    const ProgType aMode = kProportionalProgress );
     ZProgress();
     ~ZProgress();

// progress bar manipulation:
     virtual Boolean      InformProgress( const long progressSoFar );
     virtual void         SetDelay( const short delayTicks );
     virtual void         SetMessage( const Str255 aMsg );
     virtual void         SetMessage( const short resID, const short index);

     virtual void         SetUp();
     virtual void         ClickItem( const short theItem );
     virtual void         Type( const char theKey, const short modifiers );
     virtual void         SetMode( ProgType aMode );

     inline    long       GetEstimatedCompletion() { return estTicksToFinish; };
     virtual void         SetEventRatio( short aRatio );

protected:
     virtual void         EstimateCompletionTime();
     virtual void         UpdateTimeToComplete();
};
```

Data Members

<**theType**> is the type of progress dialog, stop or cancel.
<**createTime**> is the time when the dialog was created

<**deferTime**> is the amount of time to wait before showing the dialog, in ticks.
<**estTicksToFinish**> is the estimated number of ticks to go before the dialog will complete
<**fAbort**> is set to TRUE by the cancel button, which will stop the process.
<**evtProcRatio**> and <evtProcCount> are used to determine how often to handle events while processing. These allow you to fine tunethe amountof time given up to other processes while you run. Usually the default values will be fine.

Methods

```
ZProgress( ZCommander* aBoss,
           const short dialogID = kStdProgressResID,
           const long maxValue = 100,
           const short pType = kCancelType,
           const ProgType aMode = kProportionalProgress );
```

Constructor sets everything up ready to go. Unlike other dialogs, it is NOT necessary to call InitZWindow for this particular dialog- the constructor does that since this is not usually subclassed. <aBoss> is the dialog's boss, usually a window or gApplication. <dialogID> is the DLOG id that defines the layout of the dialog. MacZoop is supplied with standard resources for this, which are used by default. You can use any other dialog you want, provided the items numbers are the same. <maxValue> is the final value of the progress operation- set this to the terminal count of your loop, etc. <pType> is the basic type of the dialog. It can be one of:

```
enum
{
     kCancelType,
     kStopType,
     kNoButton
};
```

This sets the wording of thecancel button, or hides it altogther, depending on what you want. If hidden, the operation can't be cancelled by the user. <aMode> sets whether the progress bar is displayed as a proportional bar or a "barbers pole" stripe.

```
virtual Boolean      InformProgress( const long progressSoFar );
```

Call this method regularly during your lengthy operation, providing a value relative to the original <maxValue> about how far you've got. This method updates the bar as necessary, shows the dialog after the initial set delay, and occasionally processes events. It also handles the user's clicks on the cancel button. In short, it does everything.

```
virtual void         SetDelay( const short delayTicks );
```

Sets the initial delay before the dialog will show up. It is useful to set this to a couple of seconds- that way if your process can vary in length, the user won't be bothered by the dialog if the process completes quickly, but will see it if it's likely to take a while.

```
virtual void         SetMessage( const Str255 aMsg );
virtual void         SetMessage( const short resID, const short index);
```

Sets the message displayed above the progress bar to the string passed, or sets it from a STR# resource with the resID and index passed. You can use this message for any purpose you wish.

```
virtual void        SetUp();
virtual void        ClickItem( const short theItem );
virtual void        Type( const char theKey, const short modifiers );
```

These override ZDialog's methods to intercept various events and activities. You do not need to call or care about these.

```
virtual void        SetMode( ProgType aMode );
```

Changes the progress bar mode on the fly between proportional and striped. In striped mode, the value is still retained so you can flip between modes if you want.

```
inline    long      GetEstimatedCompletion() { return estTicksToFinish; };
```

Returns the estimated time it will take, in ticks, for the operation to complete, based on how long it's taken so far.

```
virtual void        SetEventRatio( short aRatio );
```

Sets how many calls to InformProgress() are required before an event is processed. This defaults to 3. By setting a larger number, your process will go slightly faster, but others will go slower. A smaller number has the inverse effect. Note that if you do not process events frequently enough, the user will perceive the interface as responding very slowly, since the cancel button, etc requires event processing to operate.

```
virtual void        EstimateCompletionTime();
virtual void        UpdateTimeToComplete();
```

Internal methods compute the time remaining and update the "Time remaining:" text respectively. The time is estimated based on a linear extrapolation from the time taken so far. If your process proceeds steadily, it should be accurate, but if erratic the time estimate can be off by quite some margin.

Comments

Appearance is used where available. If not, the progress bar is drawn using some standard colour patterns which give the same appearance as the System 7.x Finder.

To use this class, you need to add ZProgress.rsrc and ZPBarDialogItem to your project. Note also that if you turn off _DIALOG_EXTENSIONS, this object won't work properly, though nothing bad should happen. If you replace the resources with your own, be careful to keep the item numbers of the standard items the same.

You can add a progress bar control to any dialog using the ZPBarDialogItem class, but in that case you have to maintain the value of the indicator yourself.

Class Name: ZPBarDialogItem	Based on: ZDialogItem
Type:	Framework Class
Description:	Dialog item displays a progress bar

This is a custom dialog item that displays a progress bar. It is Appearance aware where available, or can mimic the System 7.x progress bar style if not.

Class Definition

```
class     ZPBarDialogItem : public ZDialogItem
{
protected:
     Rect           pRect;              // bar rect
     long           value;              // current value
     PixPatHandle   stripesPat;         // stripes pat
     ProgType       displayMode;        // current mode
     short          forePatID;          // ID of 'ppat' for bar
     short          backPatID;          // ID of 'ppat' for background of bar
     short          stripesPatID;       // ID of 'ppat' for striped pattern

public:
     ZPBarDialogItem( ZDialog* aDialog, const short item );
     ZPBarDialogItem();
     ~ZPBarDialogItem();

     virtual void         InitItem( const short paramCount, const long params[] );
     virtual void         DrawItem();
     virtual void         SetValue( const long aValue );
     virtual long         GetValue() { return value; };
     virtual void         SetMaximum( long aMax ) { maxLim = aMax; };
     virtual void         SetDisplayMode( ProgType aMode );
     virtual void         SetBounds( Rect* newBounds );

     inline    ProgType   GetDisplayMode() { return displayMode; };

protected:
     virtual void         CycleStripe();
     inline    void       DrawEmptyBar();
     inline    void       UpdateBar();
};
```

Data Members

<**pRect**> is the current filled portion of the bar
<**value**> is the current value of the bar
<**stripesPat**> is the pattern used for the “barbers pole” stripe
<**displayMode**> is the mode of thebar- proportional or “barber’s pole”
<**forePatID**> i s the ID of the ‘ppat’ used to draw the filled potion of the bar
<**backpatID**> is the ppat ID used for the background of the bar
<**stripesPatID**> i s the ppat ID used for the “barber’s pole” pattern

Methods

Constructor creates thebasic dialog item object. Themain set up is done by InitItem, as for all ZDialogItem derivatives.

```
virtual void        InitItem( const short paramCount, const long params[] );
```

Initialises the item from the “magic string” parameters. They are:

param 0- display mode. 0 = proportional, 1 = indeterminate (“barber’s pole”).
param 1- initial maximum value, default = 100.
param 2- background pattern resource ID (default is 128)
param 3- foreground pattern resource ID (default 129)
param 4- stripes pattern resource ID (default 130)

Note that the patterns are only used if Appearance is not available.

```
virtual void        DrawItem();
```

Draws the item to respond to an update event.

```
virtual void        SetValue( const long aValue );
```

Sets the progress bar’s value, and draws the bar to the correct proportion relative to the maximum value. If the bar is in indeterminate (stripes) mode, the animation is cycled and the bar redrawn.

```
virtual long        GetValue();
```

Returns the current value.

```
virtual void        SetMaximum( long aMax );
```

Sets the maximum value of the progress bar (the value at which it will appear full).

```
virtual void        SetDisplayMode( ProgType aMode );
```

Sets the display mode to proportional or indeterminate. The current value is maintained in the indeterminate mode- you can flip between modes as you wish.

```
virtual void        SetBounds( Rect* newBounds );
```

Moves or resizes the bar as required (usually when parent window resized).

```
inline    ProgType  GetDisplayMode();
```

Returns the current display mode of the bar.

```
virtual void        CycleStripe();
```

Animates the stripes pattern by shifting its rows up one and moving the top row to the bottom. Tis is called as needed to animate thye “barber’s pole” stripes when appearance is not being

used. (Appearance animates the stripe using its own techniques).

```
inline    void      DrawEmptyBar();
```

Draws the basic outline and background of the bar (when Appearance not used). Called internally.

```
inline    void      UpdateBar();
```

Draws the bar’s parts as needed. Called by DrawItem() when Appearance not used.

Comments

When Appearance is available, this item is actually implemented as a control using the progress bar CDEF that is part of appearance. In this case, the <ctrl> bit of iType is set as well as the <kCustomDialogItemType>bit, and the control handle is stored in <macItemHandle>. If Appearance is unavailable, this item is NOT a control. Instead the progress bar is drawn directly using colour patterns. The default colour patterns supplied mimic the appearance of progress bars in the 7.x Finder.

The “magic type” for the item is ‘$$PBAR’.

You must have _DIALOG_EXTENSIONS on to use this class.

Class Name: ZListDialogItem		Based on: ZDialogItem
Type:	Framework Class	
Description:	Dialog item displays a List Manager list in a dialog	

This dialog item is used to display a list box in a dialog. It is partly based on MList, so shares the list handling API with ZLMListWindow. In addition, it can be set up using a 'LIST' resource.

Class Definition

```
class     ZListDialogItem : public ZDialogItem, public MList
{
public:

     ZListDialogItem();
     ZListDialogItem( ZDialog* aBoss, const short anID );

// initialisation
     virtual void         InitItem( const short paramCount, const long params[] );

// standard commander type stuff
     virtual void         Type( const char theKey, const short modifiers );

// appearance
     virtual void         Enable( Boolean useHistory );
     virtual void         Disable( Boolean useHistory );
     virtual void         DoHilightSelection( Boolean hiliteIt );

// mousing around
     virtual void         Click( const Point mouse, const short modifiers );

// override these to allow us to set up item colours, fonts, etc
     virtual void         MLSetSelection( Cell aCell );
     virtual void         MLSetCell( Cell theCell, Ptr buf, short length );
     virtual short        MLAppendRow();
     virtual short        MLAppendCol();
     virtual short        MLAppendRowInAlphaOrder( Str255 s );

     virtual void         SetBounds( Rect* newBounds );

protected:
     virtual void         DrawItem();
     virtual void         MLNewCellSelected( Cell newCell );
     virtual void         MLCellDoubleClicked( Cell theCell );
};
```

Data Members

none.

Methods

Constructor creates basic object as a dialog item + MList object. Nothing remarkable about it.

```
virtual void        InitItem( const short paramCount, const long params[] );
```

Initialsises the item from the “magic string” parameter list. The parameters are:
param 0- resource ID of ‘LIST’ resource to initialise the item
param 1- resource ID of a ‘STR#’ resource to preload strings from, but only if LIST ID is 0.

This method creates the ist manager list and initialisese it from the LIST resource, or if this parameter is 0 or missing, to a 1-column list preloaded with strings from the STR# resource.

```
virtual void        Type( const char theKey, const short modifiers );
```

Allows typing when the item has the focus to be passed to MLKeyNavigation(). This actions the arrow keys, command-arrow keys and typing of the first few letters of an item selects it, if thelist is sorted alphabetically.

```
virtual void        Enable( Boolean useHistory );
virtual void        Disable( Boolean useHistory );
```

Handles the enabling and disabling of the item by activating and deactivating the list.

```
virtual void        DoHilightSelection( Boolean hiliteIt );
```

Responds to the focus status of the item by activating and deactivating the list’s scrollbars. Not ethat the List Manager does not allow a distinction between the enabled state of the list and the active state, so we do our best to give a meaningful behaviour in our dialog, which does distinguish the two. In any case, it’s useful to show the selection in the list even if the item does not have the focus- that way the user can see what is selected at all times, usually useful in a dialog.

```
virtual void        Click( const Point mouse, const short modifiers );
```

Passes mouse clicks on to MLClick, after focusing the item.

```
virtual void        MLSetSelection( Cell aCell );
virtual void        MLSetCell( Cell theCell, Ptr buf, short length );
virtual short       MLAppendRow();
virtual short       MLAppendCol();
virtual short       MLAppendRowInAlphaOrder( Str255 s );
```

Thsee are all overidden methods of MList. They pereform the same function, but in addition preserve and set up the correct drawing state for the dialog item. Thus as data is added to the list, it is displayed in the correct font and colours for the item.

```
virtual void        SetBounds( Rect* newBounds );
```

Allows the item to be moved and resized when the parent dialog is resized.

```
virtual void        DrawItem();
```

Updates the item using MLUpdate.

```
virtual void        MLNewCellSelected( Cell newCell );
virtual void        MLCellDoubleClicked( Cell theCell );
```

Overrides of MList which simply send messages to the boss, which can act on them as required.

For example, it’s common for a double-click on a list item to act as a shortcut for a button in the dialog. However, the dialog itself must perform this link.

Comments

Please refer to the full description of MList for other list operations you can perform using this item. This API is common to this item and ZLMListWindow.

This item is set up to be able to take the focus when the user click in it, or tabs to it. However, one of the flags in the LIST resource sets this, so there is a way to avoid the item becoming the focus. In this case mouse clicks will not be processed.

The “magic type” for this item is ‘$$LIST’.

You must have _DIALOG_EXTENSIONS on to use this class.

ZListDialogItem messages

These are:

```
enum
{
     msgNewListItemSelected = 'lstc',
     msgListItemDoubleClicked = 'lsdb'
};
```

Message data is a pointer to the cell that was selected or double-clicked.

Class Name: ZIconListDialogItem		Based on: ZListDialogItem
Type:	Framework Class	
Description:	Dialog item displays a List Manager list in a dialog, with a custom LDEF that displays a series of icons.	

This is a dialog item that diaplays an array of icons using a custom LDEF (built-in). It can form the user-interface for a multi-part dialog, using the icons as a selector, for example. It inherits most of its behaviour from ZListDialogItem. It's a good illustration of how to build on existing classes to create custom dialog items.

Class Definition

```
class     ZIconListDialogItem  : public ZListDialogItem
{
protected:
     Boolean         showTitles;     // show icon titles?
     Boolean         smallIcons;     // TRUE if we use 16 x 16 rather than 32 x 32
     IconHilite      iHilite;        // how to hilite the icons

public:

     ZIconListDialogItem();
     ZIconListDialogItem( ZDialog* aBoss, const short anID );
     ~ZIconListDialogItem();

// initialisation
     virtual void    InitItem( const short paramCount, const long params[] );

     virtual void    MLDraw1Cell( Rect* area, Cell theCell, Boolean cellHilite );
     virtual void    MLHilite1Cell( Rect* area, Cell theCell, Boolean cellHilite );
     virtual void    AppendIcon( IconInfo* info );

protected:
     virtual void    AddIconCells( IconListBoxHdl lbH );
     virtual void    DisposeIconCells();
};
```

Data Members

<**showTitles**> is TRUE if the icons have titles
<**smallIcons**> is TRUE if the icons are small
<**iHilite**>is a set of flags indicating how to hilite selected icon cells.

Methods

```
     virtual void    InitItem( const short paramCount, const long params[] );
```

Initialises the item from the "magic string" parameter list. The parameters are:
param 0- resource ID of 'LIST' and 'ICLB' resource used to set up the list and the icons within it. If 0 or missing, the list is set up as a 1-column list with no items initially- you can append icons programmatically using AppendIcon.If you want a horizontal list or other arrangement, you have to set up the required resources. This creates the list and sets it up, installing the custom LDEF callback.

```
virtual void    MLDraw1Cell( Rect* area, Cell theCell, Boolean cellHilite );
```

Overrides the cell drawing method of MList to draw icons rather than the usual strings. The icons are drawn according to the style set up in the ICLB resource.

```
virtual void    MLHilite1Cell( Rect* area, Cell theCell, Boolean cellHilite );
```

Overrides MList to hilite icons according to the ICLB settings as the user intercats with the item.

```
virtual void    AppendIcon( IconInfo* info );
```

Appends an icon to the list programmatically. This should only be used with a 1-column list (icon is appended as last row of first column).

```
virtual void    AddIconCells( IconListBoxHdl lbH );
```

Sets up the list data from the original ICLB resource. Do not call or use directly.

```
virtual void    DisposeIconCells();
```

Disposes ofthe icon data when the list is disposed. Do not call directly.

Comments

Icon data is stored in the list as records of type IconInfo. This is also what you need to pass to the AppendIcon() method. The structure is defined as:

```
typedef struct
{
     IconType   iType;              // icon's type (colour, family, etc)
     Handle     theIcon;            // handle to the icon if loaded
     short      iconID;             // resource ID of the icon
     long       userData;           // reserved for your own use if needed
     Str31      title;              // icon's title string, if any
}
IconInfo;
```

This is normally set up by reading the ‘ICLB’ resource, described in the User Manual.

This object was designed originally for a single purpose, namely as a selector device for a multi-part preferences dialog. While it is easily adapted to many other applications, you may find certain things restrictive- for example the cell size can only really accommodate up to 32 x 32 icons. It may need some tweaks to work really well in other situations.

This item should normally be permitted to take the focus as required.

The “magic type” for this item is ‘$$ICLB’

You must have _DIALOG_EXTENSIONS on to use this class.

Class Name: ZCPopDialogItem	Based on: ZDialogItem
Type:	Framework Class
Description:	Dialog item displays a pop-up selector for picking colours from a fixed palette.

This class implements a useful button-type device for picking colours. A pop-up menu displays the available colours when clicked, and the user simply picks the desired one. The closed state of the button displayes the selected colour.

Class Definition

```
class     ZCPopDialogItem : public ZDialogItem
{
protected:
     short     clutID;
     short     lastIndex;
     RGBColor  lastColour;

public:

     ZCPopDialogItem( ZDialog* aDialog, const short item );
     ZCPopDialogItem();

     virtual void         InitItem( const short paramCount, const long params[] );

     virtual void         DrawItem();
     virtual void         Click( const Point where, const short modifiers );

     virtual void         GetChosenColour( RGBColor* aColour );
     virtual void         SetColour( RGBColor* aColour );
};
```

Data Members

<**clutID**> is the table of colours to display. It should have 16, 81 or 256 colours in it- other numbers of colours will probably not display correctly sincethe menu is designed to display 4 x 4, 9 x 9 or 16 x 16 colour blocks at a time in a square grid.
<**lastIndex**> is the index of the colour last chosen in the menu.
<**lastColour**> is the actual RGB colour of the item selected.

Methods

```
     virtual void         InitItem( const short paramCount, const long params[] );
```

Initialises the item based on the “magic string” parameter list supplied. The parameters are:
param 0- the resource ID of the ‘clut’ resource to display.
If the parameter is 0 or missing, the standard system colour table of 256 colours is used.

```
     virtual void         DrawItem();
```

Draws the item when an update is required.

```
virtual void        Click( const Point where, const short modifiers );
```

Handles the mouse click in the item. This highlights the button and pops up the menu and tracks it. When the user releases the mouse, the chosen colour is set and the button redrawn to show this.

```
virtual void        GetChosenColour( RGBColor* aColour );
```

Returns the colour chosen.

```
virtual void        SetColour( RGBColor* aColour );
```

Sets the colour to the given colour. This will redraw the item to reflect the new colour, and also calculate the nearest index in the current clut.

Comments

The “magic type” for this item is ‘$$CPOP’. When the user selects a new colour, the dialog will receive the following message:

```
enum
{
     msgNewColourSelected = 'cpop'
};
```

The message data is a pointer to the chosen RGBColor.

To use this class, you must also add ColourPopUp.c and PalMDEF.c to your project. These can be found in the “Goodies and Extras” folder. This object largely calls through to basic c code in those files to do the work. It also uses a custom MDEF embedded in the application to implement the menu appearance. This relies on xDEFJump.c, which is found in the “Support Code” folder. This file is also required by any list using a custom LDEF.

You must have _DIALOG_EXTENSIONS on to use this class.

Class Name: ZTextDialogItem	Based on: ZDialogItem
Type:	Framework Class
Description:	Dialog item displays a scrollable text panel containing styled text

This dialog item is a scrollable text panel in which you can display a large amount of styled text. It uses TextEdit. Unlike an edit field, this can display much more text, supports styled text, can be editable or read-only, and has its own scrollbar.

Class Definition

```
class     ZTextDialogItem : public ZDialogItem
{
protected:
     Boolean         editable;            // TRUE if editable text
     short           resID;               // res ID of TEXT/styl resource
     short           lineHeight;          // height of a line
     short           pageHeight;          // height of a "page"

public:

     ZTextDialogItem( ZDialog* aDialog, const short item );
     ZTextDialogItem();

// overrides:
     virtual void         InitItem( const short paramCount, const long params[] );
     virtual void         DrawItem();
     virtual void         Click( const Point where, const short modifiers );
     virtual void         AdjustCursor( const Point where, const short modifiers );
     virtual void         Type( const char theKey, const short modifiers );
     virtual void         Idle();
     virtual void         DoHilightSelection( Boolean hiliteIt );

     virtual void         Enable( Boolean useHistory );
     virtual void         Disable( Boolean useHistory );

     virtual void         DoCut();
     virtual void         DoCopy();
     virtual void         DoPaste();
     virtual void         DoClear();
     virtual void         DoSelectAll();
     virtual Boolean      CanPasteType();
     virtual void         UpdateMenus();
     virtual void         HandleCommand( const long theCmd );
     virtual void         HandleCommand( const short menuID, const short itemID );

     virtual void         SetValue( const Str255 strVal );
     virtual void         SetBounds( Rect* newBounds );

// original methods:
     virtual void         SetText( Handle textH, Handle styleH = NULL );
     virtual void         GetText( Handle textH, Handle styleH = NULL );
     virtual void         Scroll( short delta );
     virtual void         DoScroll( short partCode );

     virtual void         CalScroll();
```

```
     inline    TEHandle   GetTextEditHandle() { return pwMirror; };

protected:
     virtual void         MakeMacTEAndScroll();
     virtual void         PreloadText();
     virtual Boolean      PtInScrollbar( const Point mouse );
     virtual void         DrawDisabledBar();
};
```

Data Members

<**editable**> is tRUE if the user can edit the text. If FALSE, text is read-only.
<**resID**> is the ID of a 'TEXT' (and optional 'styl') resource used to set the initial text.
<**lineHeight**> is the height of a line for unstyled text
<**pageHeight**> is the height of a page.

Methods

```
     virtual void         InitItem( const short paramCount, const long params[] );
```

Initialises the item from the "magic string" paraneter list passed. Theparameters are:
param 0- resource ID of a TEXT/styl resource initially displayed in the item. If 0, the item is initially empty.
param 1- flag. 0= not editable (read-only), 1 = editable.
param 2- overall text alignment. 0 = left, 1 = centre, 2 = right.
This method sets up the TextEdit record and scrollbar. The scrollbar is stored in the <macItemHandle> data member- however, the ctrl bit of iType is NOT set- we do not want the default behaviour here. This item is able to take the focus whether editable or not.

```
     virtual void         DrawItem();
```

Updates the text when an update event is processed.

```
     virtual void         Click( const Point where, const short modifiers );
```

Handles the mouse in the view- this tracks the scrollbar, and if editable, positions the caret and selects text.

```
     virtual void         AdjustCursor( const Point where, const short modifiers );
```

If editable, sets the iBeam when the cursor is over the text.

```
     virtual void         Type( const char theKey, const short modifiers );
```

If editable, typing allows text to be entered. The return key is processed in a slightly unusual way. If this item has the focus, the return key causes a line break in the text, which is desirable. The normal behaviour in a dialog is to action the default item, but this is suppressed as long as this item has the focus. The enter key works as normal.

```
     virtual void         Idle();
```

Blinks the caret if the text is editable.

```
     virtual void         DoHilightSelection( Boolean hiliteIt );
```

Handles the focus state by enabling and disabling the scrollbar, and if the text is editable, by activating and deactivating the text selection as appropriate.

```
virtual void        Enable( Boolean useHistory );
virtual void        Disable( Boolean useHistory );
```

Handle enabling and disabling of the item by showing and hiding the scrollbar.

```
virtual void        DoCut();
virtual void        DoCopy();
virtual void        DoPaste();
virtual void        DoClear();
virtual void        DoSelectAll();
```

Handle all of these standard Edit commands if the text is editable.

```
virtual Boolean     CanPasteType();
```

Returns TRUE if the text is editable and the clipboard contains ‘TEXT’ data. Used to enable the Paste command.

```
virtual void        UpdateMenus();
```

Updates the menus. This item can handle the standard commands pertaining to Font, Style, Alignment, Colour and Size, and enables them accordingly if the text can be edited. Note- this means that the same menus used for ZTextWindow also work here without any extra coding required.

```
virtual void        HandleCommand( const long theCmd );
virtual void        HandleCommand( const short menuID, const short itemID );
```

Handle standard commands pertaining to Font, Size, Alignment, Colour and Style, if the text is editable.

```
virtual void        SetValue( const Str255 strVal );
```

Sets the text to the string passed. This is a token gesture to allow SetValue to do something, but it would rarely be used for this item.

```
virtual void        SetBounds( Rect* newBounds );
```

Allows the item to be moved or resized when the dialog is resized. Also recalibrates the text and scrollbar.

```
virtual void        SetText( Handle textH, Handle styleH = NULL );
```

Sets the text to the data passed in <textH>, replacing the existing text. If <styleH> is not NULL, the text may be styled. Recalibrates the scrollbar as required.

```
virtual void        GetText( Handle textH, Handle styleH = NULL );
```

Copies the text and optionally the style data into the handles passed. The handles are resized to accommodate the data.

```
virtual void        Scroll( short delta );
```

Scrolls the text by <delta> pixels. Called internally.

```
virtual void        DoScroll( short partCode );
```

Part of the callback to handle scrollbar tracking. Do not use this directly.

```
virtual void        CalScroll();
```

Recalibrates the scrollbar according to the bounds rectangle and the amount of text. Called internally as required.

```
inline    TEHandle  GetTextEditHandle();
```

Returns the TextEdit handle if you want to do your own thing with it.

```
virtual void        MakeMacTEAndScroll();
```

Creates the TextEdit record and scrollbar as part of the item initialisation.

```
virtual void        PreloadText();
```

Preloads the text from the original resource specified.

```
virtual Boolean     PtInScrollbar( const Point mouse );
```

Returns TRUE if the point is in the scrollbar as opposed to the text area.

```
virtual void        DrawDisabledBar();
```

Draws the scrollbar in the disabled state- his makes the item look correct when disabled because the scrollbar itself has been hidden. This is intended to give a consistent appearance in the disabled state between different dialog items.

Comments

The “magic type” for this item is ‘$$TEXT’. This class also requires that you add TextStyleUtils.cpp to your project.

You must have _DIALOG_EXTENSIONS on to use this class.

Class Name: ZScrollerDialogItem	Based on: ZDialogItem
Type:	Framework Class
Description:	Dialog item is a generic class for displaying scrollable content in a dialog item

This is a dialog item that can display any scrollable content. The actual content will be decided by your subclass. This item is analogous to ZScroller, but is in the form of a dialog item.

Class Definition

```
class     ZScrollerDialogItem  : public ZDialogItem
{
protected:
     ControlHandle        theHBar;        // horizontal scroll bar
     ControlHandle        theVBar;        // vertical scrollbar
     Rect                 scrollBounds;   // rect of area scrolled "content"
     short                hScale;         // pixels shifted per unit horizontally
     short                vScale;         // pixels shifted per until vertically
     short                cInitValue;     // used for live scrolling support
     Boolean              hasHBar;        // true if has horizontal bar
     Boolean              hasVBar;        // true if has vertical bar
     Boolean              hasGrowBox;     // true if has grow box

public:
     ZScrollerDialogItem( ZDialog* aDialog, const short item );
     ZScrollerDialogItem();

// must override to do something useful:
     virtual void    DrawContent() {};
     virtual void    ClickContent( const Point mouse, const short modifiers ) {};

// general overrides:
     virtual void    InitItem( const short paramCount, const long params[] );
     virtual void    DrawItem();
     virtual void    Click( const Point mouse, const short modifiers );
     virtual void    ClickScroll( const Point mouse );
     virtual void    AdjustCursor( const Point mouse, const short modifiers );
     virtual void    DoHilightSelection( Boolean hiliteIt );
     virtual void    Enable( Boolean useHistory );
     virtual void    Disable( Boolean useHistory );
     virtual void    HideScrollbars();
     virtual void    MoveScrollbars();
     virtual void    SetBounds( Rect* newBounds );

// unique methods:
     virtual void    SetScrollRect( const Rect& aBounds );
     virtual void    SetScrollRect( short top, short left,
                                    short bottom, short right );
     virtual void    SetScrollRect( Point tl, Point br );
     virtual void    GetScrollRect( Rect* aBounds ) { *aBounds = scrollBounds; };

     virtual void    SetOriginToScroll();
     virtual void    GetContentRect( Rect* r );
     virtual void    GetPosition( short* hPosition, short *vPosition );
     virtual void    SetScrollAmount( const short hAmount, const short vAmount );
```

```
protected:
     virtual void    Scroll( const short dH, const short dV );
     virtual void    ScrollTo( const short hPosition, const short vPosition );
     virtual void    MakeScrollbars();
     virtual void    DrawDisabledBars();
     virtual void    CalculateControlParams();

public:
     virtual void    ScrollHandler( const ControlHandle aCtl, const short partCode );
};
```

Data Members

<**theHbar**> and <**theVBar**> are the handles to the scrollbar controls.
<**scrollBounds**> is the rectangle of the scrollable area (virtual drawing space).
<**hScale**> and <**vScale**> are the amount to shift the view for each click in the scroll arrows.
<**cInitValue**> is used to track the original position when live scrolling.
<**hasHBar**> and <**hasVBar**> specify which scrollbars should be added to the window.
<**hasGrowBox**> specify if the item also displays a grow box.

Methods

```
     virtual void    DrawContent();
     virtual void    ClickContent( const Point mouse, const short modifiers );
```

These are overridable methods for drawing the content and handling the mouse within the content area, analogous to similarly named methods in ZScroller. The default methods do nothing.

```
     virtual void    InitItem( const short paramCount, const long params[] );
```

Initialises the item based on the “magic string” parameter list for the item. The parameters are:
param 0- feature flags. Bit 0= has vertical bar, bit 1 = has horizontal bar, bit 2 = has grow box.
param 1- width of virtual scrolling area
param 2- height of virtual scrolling area
This method creates the scrollbars and sets up the item.

```
     virtual void    Click( const Point mouse, const short modifiers );
     virtual void    ClickScroll( const Point mouse );
     virtual void    AdjustCursor( const Point mouse, const short modifiers );
     virtual void    DoHilightSelection( Boolean hiliteIt );
     virtual void    Enable( Boolean useHistory );
     virtual void    Disable( Boolean useHistory );
     virtual void    SetBounds( Rect* newBounds );
```

These are all standard overrides of ZDialogItem and do the usual thing. Note that this item implements the command-click drag scrolling that ZScroller does- the cursor shape is set as required.

```
     virtual void    HideScrollbars();
     virtual void    MoveScrollbars();
```

Hides and moves the scrollbars to the item’s edges when the item size changes. Analogous to similar methods in ZScroller.

```
virtual void    SetScrollRect( const Rect& aBounds );
virtual void    SetScrollRect( short top, short left,
                               short bottom, short right );
virtual void    SetScrollRect( Point tl, Point br );
```

This method sets the size of the virtual drawing area. It is analogous to the SetBounds method of ZScroller- note that because ZDialogItem already has a SetBounds method for a different purpose, and defines a data member called <bounds> for the item's boundary rectangle, it was unfortunately necessary to use a different name than ZScroller here.

```
virtual void    GetScrollRect( Rect* aBounds );
```

Returns the size of the virtual drawing area.

```
virtual void    SetOriginToScroll();
```

Sets up the grafPort's origin to allow for the scroll offset and the item's position with in the window. This allows DrawContent to draw without having to care about the port's origin- it draws with respect to the <scrollBounds> rectangle. Called as needed to establish the drawing environment.

```
virtual void    GetContentRect( Rect* r );
```

Returns the interior area of the dialog item- the visible drawable area less the scrollbars.

```
virtual void    GetPosition( short* hPosition, short *vPosition );
```

Returns the current scroll offset of the view.

```
virtual void    SetScrollAmount( const short hAmount, const short vAmount );
```

Sets the amount the view will scroll when the scroll arrows are clicked. The default scale is 10 pixels.

```
virtual void    Scroll( const short dH, const short dV );
```

Scrolls the view by dH and dV pixels. Do not call directly.

```
virtual void    ScrollTo( const short hPosition, const short vPosition );
```

Scrolls the view to the given position.

```
virtual void    MakeScrollbars();
```

Creates the scrollbars and positions them initially.

```
virtual void    DrawDisabledBars();
```

Draws the scrollbar areas when the item is disabled. The appearance is intended to match other dialog items in the same state.

```
virtual void    CalculateControlParams();
```

Computes scrollbar values when the item or drawing area size changes.

```
virtual void    ScrollHandler( const ControlHandle aCtl, const short partCode );
```

Callback method implements scrolling while the scrollbar controls are tracked. Do not use or call directly.

Comments

The “magic type” for this item is ‘$$SCRL’. It is unlikely that you will ever create an item of this type, since without subclassing it, it will not do anything useful.

ZGWorldDialogItem is a more useful subclass of this that displays a ZGWorld as its contents.

You must have _DIALOG_EXTENSIONS on to use this class.

Class Name: ZGWorldDialogItem	Based on: ZScrollerDialogItem
Type:	Framework Class
Description:	Dialog item with scrolling content that displays a GWorld

This is a dialog item derived from ZScrollerDialogItem that draws the interior using an attached ZGWorld object. This may handily be used to implement image previews in dialogs, etc (e.g. Photoshop style effects dialog that shows the effect in a preview item).

Class Definition

```
class     ZGWorldDialogItem    : public ZScrollerDialogItem
{
protected:
     ZGWorld*        itsOffscreen;

public:
     ZGWorldDialogItem( ZDialog* aDialog, const short item );
     ZGWorldDialogItem();
     ~ZGWorldDialogItem();

     virtual void         InitItem( const short paramCount, const long params[] );
     virtual void         DrawContent();

     inline     ZGWorld*   GetGWorld() { return itsOffscreen; };
};
```

Data Members

<**itsOffscreen**>- the ZGWorld object that it displays

Methods

```
     virtual void         InitItem( const short paramCount, const long params[] );
```

Initialises the item using the “magic string” parameter list. The parameters are:
param 0- feature flags. bit 0 = has a vertical scrollbar, bit 1 = has a horizontal scrollbar
param 1- Resource ID of a PICT resource to display. If 0, other parameters establish GWorld.
param 2- width of GWorld image
param 3- height of GWorld image
param 4- pixel depth of image (1, 2, 4, 8, 16 or 32)
param 5- ID of colour table to use for indexed images (0 = system colours).
This method then creates the ZGWorld object accordingly.

```
     virtual void         DrawContent();
```

Draws the interior of the item by copying the GWorld bits to the window.

```
     inline     ZGWorld*   GetGWorld();
```

Returns the ZGWorld object if you want to do something with it.

Comments

The “magic type” for this item is ‘$$GWRL’. Your project must also have ZGWorld if you use this.

You must have _DIALOG_EXTENSIONS on to use this class.

Class Name: ZHexEditor	Based on: ZScroller
Type:	High-level Class
Description:	Complete hexadecimal editor UI

ZHexEditor implements a complete hexadecimal editor. While useful in its own right, it was originally written to show how full-featured UIs can be created. It is not intended to be subclassed, though there is no reason why it could not if this made sense for your application.

Class Definition

```
class     ZHexEditor : public ZScroller
{
protected:
     Ptr             dumpArea;
     long            dumpLength;
     Handle          tempStore;
     long            selStart, selEnd;
     RgnHandle       selectionRgn;
     long            maxDisplayLength;
     short           hdStart, hdEnd;
     short           lineHeight, ascent, emSpace;
     Boolean         edited;
     Boolean         caretPhase;
     Boolean         nibPhase;
     Boolean         halfOff;

public:

     ZHexEditor( ZCommander* aBoss, const short windID );
     ZHexEditor();
     virtual ~ZHexEditor();

// window + scroller overrides:
     virtual void    InitZWindow();
     virtual void    Draw();
     virtual void    DrawContent();
     virtual void    ClickContent( const Point mouse, const short modifiers );
     virtual void    Type( const char theKey, const short modifiers );
     virtual void    AdjustCursor( const Point mouse, const short modifiers );
     virtual void    Activate();
     virtual void    Deactivate();
     virtual void    Idle();
     virtual Boolean ClickInSamePlace( const Point a, const Point b );

// standard commands:
     virtual void    UpdateMenus();
     virtual void    DoSelectAll();
     virtual void    DoCopy();
     virtual void    DoClear();
     virtual void    DoPaste();
     virtual Boolean CanPasteType();
     virtual void    HandleCommand( const long aCmd );

// comrade stuff:
     virtual void    ReceiveMessage( ZComrade* aSender, long aMsg, void* msgData );
```

```
// file I/O
     virtual void    OpenFile( const OSType fType, Boolean isStationery = FALSE );
     virtual void    SaveFile();

// setting content of the dump and selection in it:
     virtual void    SetDumpMemory( Ptr dumpThis, long dumpBytes );
     virtual Boolean DumpChanged();
     virtual void    GetDumpMemory( Ptr dumpStorage, long* dumpBytes );
     virtual void    GetDumpMemory( Handle aHandle );
     virtual void    SetSelection( long start, long end );

// general purpose search method:
     virtual Boolean FindByteString( Ptr targData,
                                     long targDataLen,
                                     long* startOffset,
                                     Boolean wrap = FALSE );

// info:
     inline void          GetSelectionRange( long* start, long* end );
     inline long          GetLength() { return dumpLength; };

protected:
     inline  void    DrawHexLine( Ptr lineStartAddr,
                                  char bytesInLine = BYTES_PER_LINE );
     inline  void    DrawLongAsHex( long val );
     virtual long    GetOffset( short hLoc, short vLoc );
     virtual void    CalcSelection( long offsetStart, long offsetEnd,
                                    RgnHandle aRgn );
     virtual void    DrawCaret( Boolean state );

// find dialog:
     virtual void    OpenFindDialog();

// scroller overrides:
     virtual void    MoveScrollbars();
     virtual void    HideScrollbars( Boolean validateArea );

// hex editor utilities:
     void            PrepareForEdit();
     void            Munge( Ptr dataToInsert, long dataLength );
     void            RecalcBoundsAndSelection();
     void            MoveCaret( short cDirection, Boolean shifted = FALSE );
     void            RedrawOnType( long offset );
     virtual void    TypeInData( char theKey );
     void            ScrollToCaret();
     void            SetSelectionRgn( RgnHandle aRgn );
     void            ConvertHexSearchString( Str255 aString );
     virtual short   CalcCaretHOffset( long offset );

     virtual void    DrawPlacard();
     virtual void    DrawHeader();
};
```

Data Members

<**dumpArea**> is a pointer to the start of the data the editor displays
<**dumpLength**> is the length of the data
<**tempStore**> is a Handle to the data when edited
<**selStart**> and <**selEnd**> are the byte offsets of the start and end of the current selection.
<**selectionRgn**> is the highlight region of the selection.
<**maxDisplayLength**> is the maximum number of bytes we can display/edit at once.
<**hdStart**> and <**hdEnd**> are the horizontal pixel positions where the actual data is drawn.
<**lineHeight**>, <**ascent**> and <**emSpace**> are font characteristics used to calculate where stuff is

drawn and where the mouse is in the data.
<**edited**> is TRUE if the user has changed the data at all
<**caretPhase**> is used to control caret blinking and visibility.
<**nibPhase**> is used to controlwhich nibble of a byte is being entered.
<**halfOff**> is used to position the caret between nibbles while editing.

Methods

Constructor initialises data members and basic scroller in the usual way.

```
virtual void    InitZWindow();
```

Initialises the whole caboodle, including underlying scroller. MUST be called after creating the object.

```
virtual void    Draw();
virtual void    DrawContent();
virtual void    ClickContent( const Point mouse, const short modifiers );
virtual void    Type( const char theKey, const short modifiers );
virtual void    AdjustCursor( const Point mouse, const short modifiers );
virtual void    Activate();
virtual void    Deactivate();
virtual void    Idle();
virtual Boolean ClickInSamePlace( const Point a, const Point b );
```

Standard overrides of ZWindow and ZScroller. As you might expect, these do the usual things, drawing the content of the window, handling the mouse clicks in the content, accepting keyboard input, adjusting the cursor, activating and deactivating the selection, blinking the caret and resolving double-clicks.

```
virtual void    UpdateMenus();
```

Enables the commands the editor can respond to, which are the usual Edit commands.In addition, we define one command of our own, which is “Find Hexadecimal”. A dialog is provided for this.

```
virtual void    DoSelectAll();
virtual void    DoCopy();
virtual void    DoClear();
virtual void    DoPaste();
```

Respond to the usual commands. Data cut or copied from the editor is placed on the clipboard as ‘TEXT’ whether it is or not. Any data on the clipboard can be pasted.

```
virtual Boolean CanPasteType();
```

Returns TRUE if there is ANY data on the clipboard (uses the wild card data type- see ZClipboard).

```
virtual void    HandleCommand( const long aCmd );
```

Responds to the Find Hex command by calling the OpenFindDialog() method. All other commands are punted.

```
virtual void    ReceiveMessage( ZComrade* aSender, long aMsg, void* msgData );
```

Listens for messages from the Find Hex dialog, and executes the search. See FindByteString below for a description of this process.

```
virtual void    OpenFile( const OSType fType, Boolean isStationery = FALSE );
```

Reads the hex data from a file. Any type of file can be opened.

```
virtual void    SaveFile();
```

Saves the current contents of the editor to a ‘TEXT’ file.

```
virtual void    SetDumpMemory( Ptr dumpThis, long dumpBytes );
```

Call this to set the memory that the editor is to display and edit. Thebuffer may not move in memory while the editor is displaying it.

```
virtual Boolean DumpChanged();
```

Returns TRUE if the data has been edited since it was first displayed in the editor.

```
virtual void    GetDumpMemory( Ptr dumpStorage, long* dumpBytes );
virtual void    GetDumpMemory( Handle aHandle );
```

Returns the contents of the editor to a buffer, or reads it into a handle. <dumpBytes> specifies how much data to read out- the actual amount read out will be the smaller of this value and the actual amount of data in the editor. The actual amount copied is returned. If <dumpStorage> is NULL, this returns the size of the data. If you pass a Handle, the handle is sized tothe size of the data and the whole dump is copied into the handle.

```
virtual void    SetSelection( long start, long end );
```

Sets the current selection to the start and end data offset passed. This recalculates and redraws the selection, scrolling the view so that it’s visible if necessary.

```
virtual Boolean FindByteString( Ptr targData,
                                long targDataLen,
                                long* startOffset,
                                Boolean wrap = FALSE );
```

Searches the contents of the editor for the data in <targData>, matching <targDataLen> bytes. The search begins at the offset passed in <startOffset>, and if found, the method returns TRUE and <startOffset> contains the offset where the found data was matched. If the data cannot be matched, the method returns FALSE. If <wrap> is true, the search wraps to the beginning of thebuffer again until the original <startOffset> value is reached, which aborts the search. This method can be called repeatedly to find the data again. It is called by ReceiveMessage when the Find Hex dialog interface is operated. Note- it only returns the found data offset. If you wish to set the selection to this, callSetSelection passing the <startOffset> and startOffset + targDataLen as offsets.

```
inline void          GetSelectionRange( long* start, long* end );
```

Returns the current selection offsets.

```
inline long          GetLength();
```

Returns the total amount of data in the editor.

```
inline  void    DrawHexLine( Ptr lineStartAddr,
                             char bytesInLine = BYTES_PER_LINE );
```

Draws a single line of hex data. Th eline is drawn based on the current pen position and the value of <hdStart>. This method is part of the general drawing operation of the editor, and is optimised for speed. Called internally as required.

```
inline  void    DrawLongAsHex( long val );
```

Draws the long value passed as a hexadecimal string at the current location. This is used to draw the address/offset portion of the display. Compilation options specify whether it draws a 6 or 8 digit address. Called internally.

```
virtual long    GetOffset( short hLoc, short vLoc );
```

Returns the data offset corresponding to the hLoc and vLoc pixel locations passed, which must be offset to allow for scrolling, etc. This is the basic method for determining where in the data buffer the mouse is when clicking and dragging.

```
virtual void    CalcSelection( long offsetStart, long offsetEnd,
                               RgnHandle aRgn );
```

Computes the selection region, based on the start and end offsets passed. The region is returned in <aRgn>, and does not explicitly allow for the scroll offset, etc- normally this is OK, since everything is drawn relative to the bounds of the scroller, which is best thought of as a fixed rectangle. The region is inverted to producethe selection highlight.

```
virtual void    DrawCaret( Boolean state );
```

Draws the caret in the given state. This takes into account the data members that control caret positioning while editing. Called internally.

```
virtual void    OpenFindDialog();
```

Creates or selects the Find Hex dialog. This is a standard user-interface to the FindByteString() method. The same dialog is shared by all instances of ZHexEditor- the one immediately behind it will respond to the “Find” command.

```
virtual void    MoveScrollbars();
virtual void    HideScrollbars( Boolean validateArea );
```

Overrides ZScroller methods to allow for the placard in the bottom left of the window, which displays the number of bytes in the dump.

```
void            PrepareForEdit();
```

Sets up the editor for editing as required. The original data is copied to the handle <tempStore> and all editing is done on this copy- the original data will not be changed unless the

GetDumpBytes method is called to copy it back by some other object.

```
void            Munge( Ptr dataToInsert, long dataLength );
```

Basic insert/replace method used by the editor. The current selection is replaced by the data pointed to by <dataToInsert>, of length <dataLength>. If the selection is zero length, the data is inserted at that point. If <dataToInsert> is NULL, the length is ignored and any selection is deleted. This method does not recompute the selection region, but will usually affect the selection end offset. Called internally to do the actual editing donkey-work.

```
void            RecalcBoundsAndSelection();
```

Recalculates the selection region and bounds rectangle of the scroller whenever data is edited. Called internally.

```
void            MoveCaret( short cDirection, Boolean shifted = FALSE );
```

Moves the caret. This is used to position the caret as typing is done, or to handle arrow keys and shift-arrow keys. Called internally.

```
void            RedrawOnType( long offset );
```

Causese the display to be redrawn in response to entering data at <offset> by typing. This calculates the minimum area that needs to be redrawn and does the drawing. Called internally during typing.

```
virtual void    TypeInData( char theKey );
```

Handles the entering of data from the keyboard. The character must be a valid hexadecimal digit. The editor keeps track of which nibble is actually being entered, munges the data as required and updates the display, selection and bounds rect as needed. Called internally by the Type method for actual data entry keys (arrow keys are handled by Type() itself).

```
void            ScrollToCaret();
```

Scrolls the view so that the caret is visible. This is done while typing so theuser can always see what they are entering.

```
void            SetSelectionRgn( RgnHandle aRgn );
```

Sets the selection region to the region passed, redrawing the region. This actually calculates the Xor of the new region and the old one, and inverts the difference. If the window is inactive, this calculates the outline region. Called internally as needed.

```
void            ConvertHexSearchString( Str255 aString );
```

Converts a string containing hexadecimal numbers to the actual data they represent, which is returned in the string. For example, “\p0A 12 44” is converted to the bytes 10, 18, 68 (decimal). White space is removed. This is done to convert data entered into the Find Hex dialog to something that FindByteString can actually use.

```
virtual short   CalcCaretHOffset( long offset );
```

Very low-level utility used widely. This returns the horizontal pixel offset within a single line of the display of the offset passed. This takes into account how data is grouped, the font into and the number of bytes displayed on each line. Called internally.

```
virtual void    DrawPlacard();
```

Draws the placard, which displays the number of bytes in the editor.

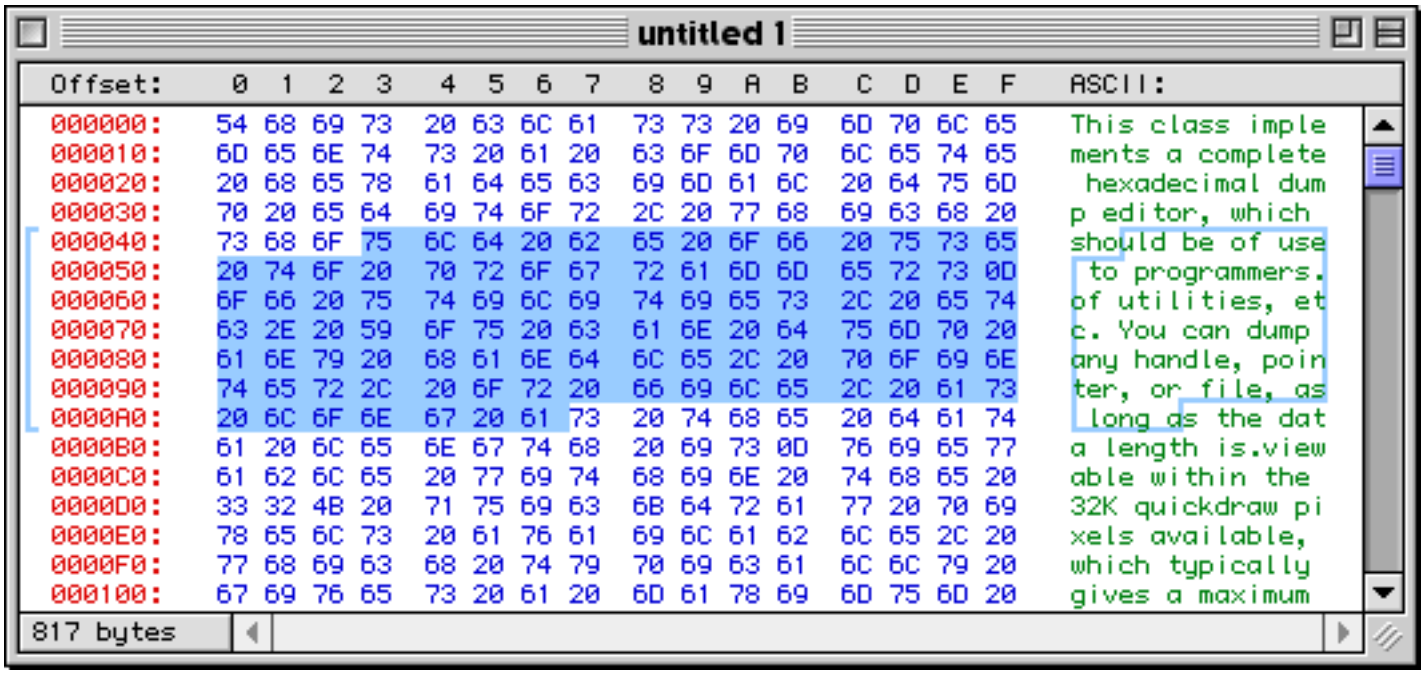
```
virtual void    DrawHeader();
```

Draws the header, which provides column headings for the various areas.

Comments

One limitation of this class is that it can't edit all that much data at once- about 40K max. The reason is that the QuickDraw space needed to display enough rows exceeds the 32,768 maximum. The best way to work around this is to convert the display to "long" coordinates, and map the displayed area onto some part of the QuickDraw plane. It would also be necessary to scale the scrollbar values so that they were not dealing with pixel offsets, as they do by default, but perhaps line count, etc. This is left as an exercise- it's not that hard to do, but was felt not worth it for the purpose for which this class was originally designed. In any case, editing more than 40K of data using a hex editor is not really a very sensible thing to do!

Here's what the editor looks like:

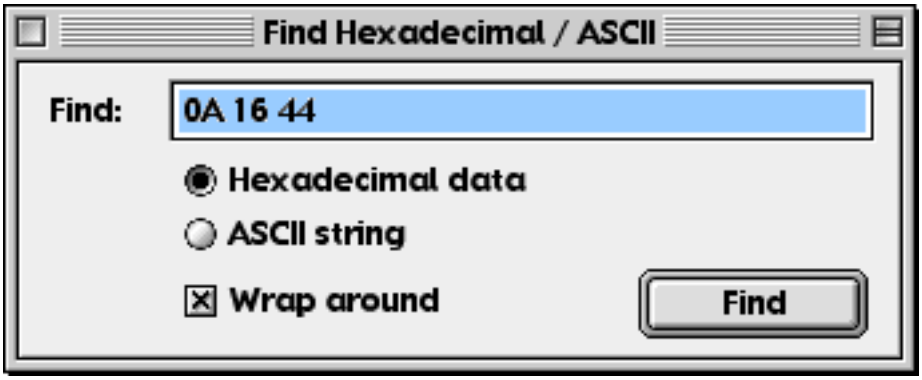


To use this class, you also need to add **HexEditor.rsrc** to your project, which contains the Find Hex dialog resources, etc. The default ID for this dialog is 199.

The Find Hex command is defined:

```
enum
{
     kCmdFindHexadecimal  = 199L
};
```

Here's what the supplied dialog looks like:



Compilation options

You can compile this editor in a variety of ways. The default options give a general purpose display as shown. Here are the various switches:

```
#define          BYTES_PER_LINE                  16
```

Sets the number of bytes displayed per line. May be set to 8 or 16. Other values not sensible probably.

```
#define          HD_USE_COLOUR                   1
```

If 1, the display is drawn in colour as shown. If 0, the display is in black. It's actually faster in black, but you'll probably only notice the difference on older Macs.

```
#define          SHOW_ABSOLUTE_ADDRESS           0
```

If 1, the offset shows the actual address of the data, not the offset from the beginning.

```
#define          USE_24_BIT_ADDRESS              1
```

If 1, the address/offset shows 6 digits, if 0, it shows 8.

```
#define          ASCII_OFFSET                    20
```

The gap to leave between the end of the dump line and the start of the ascii line, in pixels.

```
#define          COLUMN_GROUP_SIZE               4
```

The number of bytes to group into a column. 4 is typical, could be 2 or 8 or possibly more. Has not been widely tested.

```
#define         CARET_WIDTH                         1
```

Sets the pixel width of the caret. Usually 1, could be 2. More is probably not a good idea.

```
#define         SIZE_PLACARD                    1
```

Sets whether the placard is shown- usually 1, set to 0 to disable it.

```
#define         PLACARD_WIDTH                   80
```

Sets the pixel width of the placard. 80 is about right.

Class Name: ZPrinter		Based on: ZObject
Type:	Framework Class	
Description:	Helper class to ZApplication, handles basic print loop.	

ZPrinter is a helper object of ZApplication that handles the Print and Page Setup commands. It interacts with ZWindow to render window content to the printer.

Class Definition

```
class     ZPrinter   : public ZObject
{
protected:
     ZWindow*   printWindow;
     THPrint    macPrintH;
     TPPrPort   printPort;
     Rect       paperRect;
     short      pH, pV, pageStart, pageEnd, pageNum;
     Boolean    printInited;

public:
     ZPrinter( ZWindow* aWindow = NULL );
     virtual ~ZPrinter();


     virtual void    Print();
     virtual void    Print( ZWindow* aWindow );
     virtual void    PageSetUp();

     virtual void    GetPaperRect( Rect* aRect );

     inline THPrint  GetMacPrintRecord() { return macPrintH; };
     virtual void    SetMacPrintRecord( THPrint pRec );
     virtual void    SetDefault();

protected:

     virtual void    PrintLoop();
     virtual void    SetUpDocTitle( ZWindow* aWindow );

     WindowPtr  fakeWindow;
};
```

Data Members

<**printWindow**> is the window being printed.
<**macPrintH**> is the Mac toolbox print record for the print job in hand
<**printPort**> is the grafPort being drawn to.
<**paperRect**> is the paper size currently set.
<**pH**>, <**pV**>, <**pageStart**>, <**pageEnd**> and <**pageNum**> are all used to track the page range and progress while printing.
<**printInited**> is TRUE if the printer has been set up.

Methods

Constructor creates the printer, associating it with a window if desired. Note- you can have one ZPrinter per window, but MacZoop is not usually set up like this. Instead, a single ZPrinter is created that handles all print jobs. You might change this if you wanted to have a different Page Setup per window.

```
virtual void    Print();
```

Prints a previously set up window (if one was passed in the constructor). It's more usual to use:

```
virtual void    Print( ZWindow* aWindow );
```

Which prints the contents of the window passed. This method handles the complete Print command from start to finish, calling other methods to do some of its work. It also sets up a dummy window while printing (which is kept hidden) which is set to the name of the document. This prevents Print drivers from picking up the wrong title in a floating window environment.

```
virtual void    PageSetUp();
```

Handles the Page Setup command, displaying the dialog and setting up the print record and paper rect.

```
virtual void    GetPaperRect( Rect* aRect );
```

Returns the paper rectangle.

```
inline THPrint  GetMacPrintRecord();
```

Returns the print record.

```
virtual void    SetMacPrintRecord( THPrint pRec );
```

Sets the print record to the one passed, or, if NULL, sets the default. These methods can be used to get and set the print record from a resource, such as a 'PREC' saved with a particular file. This is left to your app however.

```
virtual void    SetDefault();
```

Sets the default for the print record from the Printer Driver. Called as needed when the printing is set up.

```
virtual void    PrintLoop();
```

Does all the work! This displays the Print dialog, and if the user goes ahead, the window is asked to paginate its contents to the paper rect. Once the number of pages is known, the main loop calls the window's PrintOnePage method as necessary to render the image to the printer port. The Printer Driver does the rest, and printed output should appear!

```
virtual void    SetUpDocTitle( ZWindow* aWindow );
```

Sets up the document title using the hidden window.

Index

N

O

P

Q

R

S

T

U

V

W

Z