Summit ’98

Resources

by Steven Willis

1. The Background of Resources

There is some basic information which all 4th Dimension (4D) programmers should know already about resources. This information is presented here basically as a reminder for those that have not worked directly with resources in a while. It can be useful, also, for 4D programmers which are coming directly from the Windows platform and may not be as familiar or comfortable with resources.

What are resources?

Resources are binary objects with a structured format. Resources typically contain data, including strings, numbers, pictures, icons, code, etc.

This is very similar to what many objects in 4D are like. For example, a variable of type integer in 4D has a defined structure of being a signed integer value with a length of two bytes. Another example is an array of longints. An array of longints in 4D has a particular defined structure of a series of size+1 elements, each element being a single longint.

The structural definition of resources is not limited to numeric values. Just like in 4D, there are many predefined "structural element types" (known as "resource field types") which encompass most any object structure that could ever be needed. And, again like 4D, other structural element types can be defined by a programmer (more on this in the section *Resource Field Types*, below).

Where are resources stored and what is a resource fork?

Under the MacOS, documents can have one or both of two different "sections". These sections are commonly known as "forks". For every document, there can be a data fork and a resource fork (see figure 1.1). Access to the data and resource forks in documents is handled by Toolbox calls in the MacOS. This provides for a consistent mechanism to manipulate the contents of both forks. 4D provides a layer of commands above the MacOS Toolbox for manipulating the contents of both forks of a document.

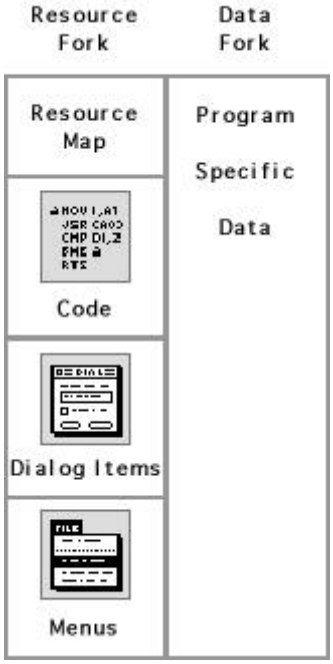

figure 1.1, the two different forks of a MacOS document.

* Image courtesy of David Adams and Dan Beckett, reprinted from “Programming 4th Dimension, The Ultimate Guide” with permission, 1998.

Under Windows, there is no predefined document structure which can encompass the two forks existing on the Macintosh. 4D handles this shortcoming by storing each MacOS document fork in a separate document. Quite often, the resource fork of a document under Windows will have an extension of ".RSR" (see figure 1.2).

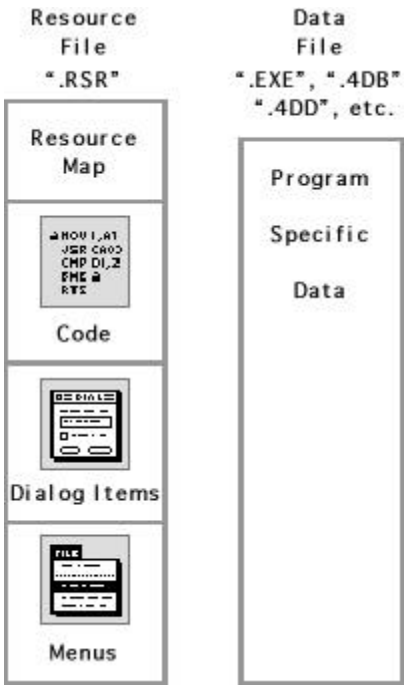

figure 1.2, how 4D handles the two forks of a document under Windows.

This is the only platform dependent issue that must be handled when using resources in 4D, as the commands in 4D to open resource documents use the path to the resource document to open. Under the Mac OS, the path to the document who's resource fork is meant to be opened is reference; under Windows, to path to the actual resource document is referenced.

Resources are stored in the resource fork of a document. In 4D, the location of the resource fork may be in a separate document (i.e. while under Windows). The resource fork of a document stores all of the resources for a particular document. Resources of different types and multiple resources of the same type may be stored in the resource fork of a document (see the section *The Basics of Understanding Resources*, below).

What is ResEdit and Resorcerer?

ResEdit is a resource editor application under the MacOS. It is available for free from Apple. It can be downloaded from a variety of sources, including the 4D Zine web site (select the Freeware link, and do a search for "ResEdit") at:

http://www.fourdzine.deepskytech.com/

ResEdit allows for the basic editing of resources. The types of resources which are supported, as well as the resource field types supported, is somewhat limited. But, overall, it is a simple and convenient starting point for beginning to work with and understand resources.

Resorcerer is an advanced resource editor for the MacOS. It provides a much broader support of resource types and resource field types. It's editors are more easily configurable and support a wider variety of data types.

Resorcerer is a commercial product available from Mathemaesthetics, Inc. Information about this product can be found at:

http://www.mathemaesthetics.com/

2. The Basics of Understanding Resources

To program effectively using resources, there are some basic concepts which must be understood. These concepts are few in number and are very easy to understand. Together, these concepts form the basis for a very powerful tool in any application development environment: resources.

What are resource types?

The resource fork of a document often contains many different resources. Each of these resources can store a variety of data structured in variety of formats.

Each resource is classified as having a particular type. Resources that share the same type share the same data structure (see the section *What are template resources?*, below).

Resource types are designated by a four byte value. Typically, this four byte resource type is referred to by its four character ASCII equivalent (i.e. string of length four); though in actuality this value is better represented by a longint in 4D. Some common resource types are (without the apostrophes):

'STR#'	1398034979	string list;
'TEXT'	1413830740	text resource;
'PICT'	1346978644	picture resource;
'clut'	1668052340	color lookup table.

When talking about resources, it is much easier to refer to resources by their ASCII equivalents. But, remember, resource types are actually referred to by a four byte value. This means that when referring to resource types by the ASCII equivalents, the case and diacritical marks of the resource type do matter. In other words, the resource types 'TEXT', 'Text', and 'text' all refer to different and distinct resource types. It is always best to refer to resource types by a longint in 4D.

What are resource names and IDs?

Each resource has values associated with it. Two of these values are the resource name and the resource ID.

The resource name is a string of up to 255 characters. It does not have to be unique, even within a particular series of resources of the same type. It is not required that there even be a name for a resource; a resource name can be empty. Usually, resource names are used to describe and identify resources to the programmer. Resource names can be used for finding resources of a particular type.

The resource ID (or resource number) is a unique identifier for a resource of a particular type. There can not be resources of the same type sharing the same resource ID. Though, resources of different types can have the same resource ID. The range of resource IDs is a signed two byte number (from -32767 through 32767, an integer in 4D).

It should be noted that certain ranges of resource IDs are typically reserved for different parts of the operating system and application space. Figure 2.1, below, shows the ranges which affect programming in 4D. In general, always use resource IDs greater than 15000 for your application development in 4D.

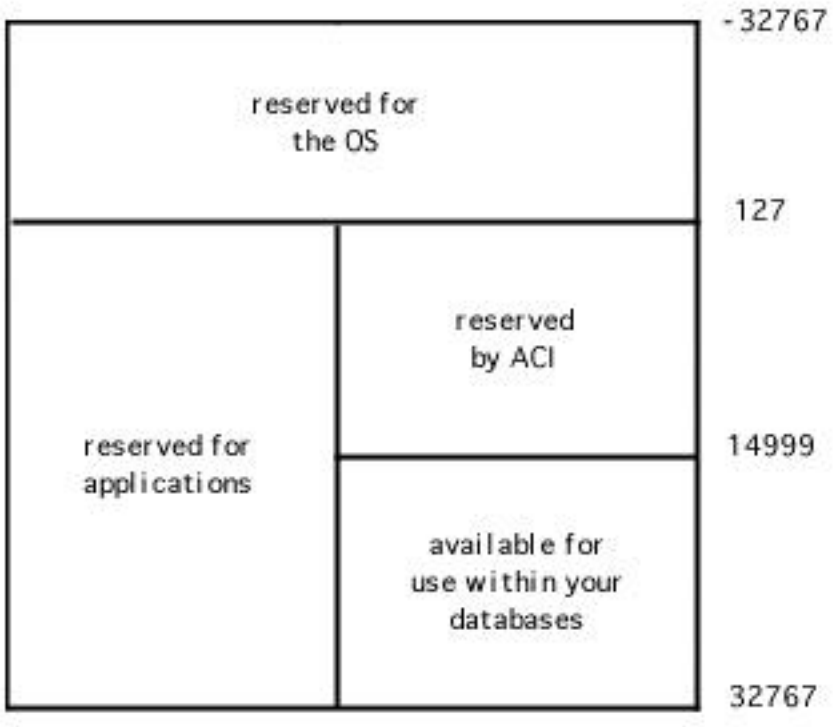

figure 2.1, table of resource ID ranges

What are resource properties?

There are other attributes which are associated with each resource. Commonly, these remaining attributes are called resource properties.

It is beyond the scope of this document to describe resource properties. Quite often resource properties are not really used a great deal by 4D programmers.

It is worth noting, though, that the resource properties of resources in application and in the operating system should not be changed. Resource properties that are changed in these locations can produce unexpected results.

For a basic explanation of resource properties, refer to the documentation that comes with 4D. For a thorough explanation of resource properties, refer to the appropriate Apple documentation for resources.

In most cases, to a 4D programmer, none of the custom resource bits should ever be set for resources which are created. The only exception is the purgable resource property bit. This resource property should be set for almost all custom resources created by a 4D programmer.

What is the resource hierarchy?

When a particular resource is referenced, by resource ID and type or by resource name and type, there are two different ways to indicate where the resource is located. One form is to actual indicate which document contains the resource being referenced. The other form is to not indicate any document in which to find the resource. This latter form is where the resource hierarchy, also known as the resource files chain, becomes a factor.

The resources in any open resource fork of a document may be used. Obviously, the structure file which is in use has a resource fork, and 4D opens this resource fork and makes it available in 4D once a database is launched.

The 4D application which is running also has a resource fork which is open and usable. And, the operating system (under the MacOS only) has a resource fork which is available.

When no particular document is indicated in a call to a resource command, the resource hierarchy is used to help determine what document to operate on (to read the resource from, to get the resource properties of, etc.). The resource hierarchy is basically a stack of the open resource forks of documents. As a resource fork is opened, it is put onto the top of this stack. And, when a resource fork is closed, it is removed from this stack and discarded. Figures 2.2, 2.3, 2.4, and 2.5 show typical resource stacks under different conditions within 4D. The stack is traversed from the top to find any resources which are referenced without specific document indicators.

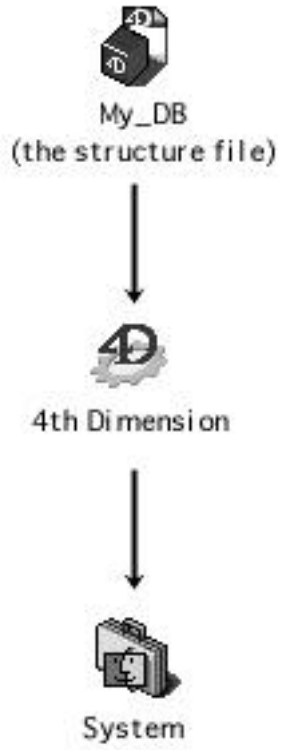

figure 2.2, a typical resource fork/file stack while developing with 4D single user under MacOS

figure 2.3, a typical resource fork/file stack while developing with 4D client/server under MacOS

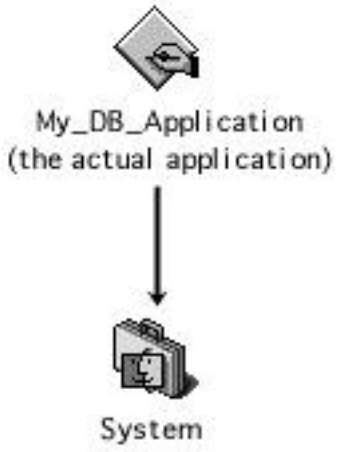

figure 2.4, a typical resource fork/file stack while deployed with a 4D database merged with 4D Engine under the MacOS

figure 2.5, a typical resource fork/file stack while developing with 4D single user under Windows

For instance, assume a call to read a resource of type 'STR#', ID 512 is made, and no particular resource fork is indicated in the call. In a standard situation in 4D (figure 2.2), the specified resource will be looked for first in the structure file. If the resource is not found, the 4D application will then have its resource fork scanned for the particular resource. If it is still not found, then the resource fork of the System file will be scanned. The location which actually has the resource will be the one which is used for the call. If the specified resource is not found anywhere in the resource hierarchy, then the call will typically return an error result of some sort.

Quite often, in 4D, it is easiest to always reference a particular document in resource calls. This assures that the desired resource fork in the intended document will be operated on. But, this must not always be the case...

What are template resources?

Each resource type has a particular, well-defined structure. This structure determines what data can be stored in each resource, how it is stored in each resource, and how much data can be stored in each resource.

The structure for many common resource types is already known and defined. The definitions of the structures of these resources are stored in a particular resource of type 'TMPL', known as template resources. Figure 2.6 shows a typical template resource.

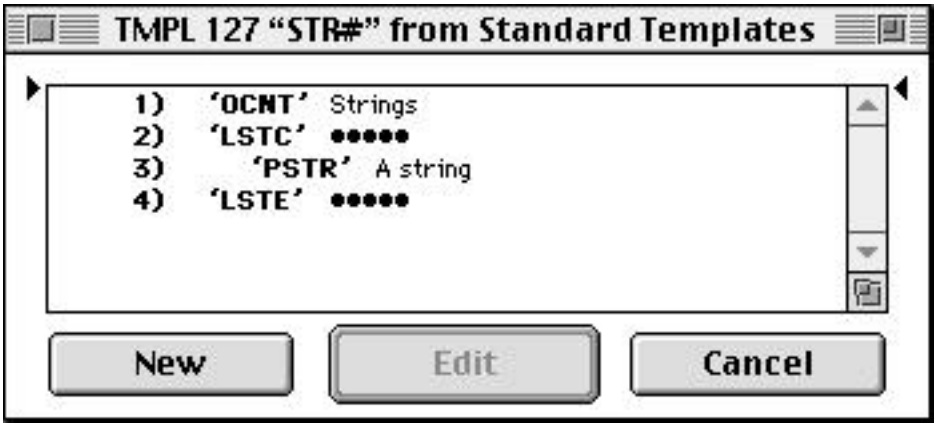

figure 2.6, a typical template resource (for 'STR#')

A template resource is just a list of paired items. The first item is a four byte value indicating the resource field type (see the section, *What are resource field types*, below). The second item is a string typically holding a field name or descriptor.

The resource name of a resource of type 'TMPL' is used to indicate what type of resource it describes. The first four bytes of the resource name indicates directly the resource type which is being described in the resource template. The resource template shown in figure 2.6 has a name of "STR#"; obviously, this template resource describes the structure of resources of type 'STR#'.

For the structure of a resource of type 'TMPL', see the section *What is the structure of a template resource*, below.

What are resource field types?

Data of many different types can be stored in a single resource. It is common practice to store, for instance, a pair of four byte unsigned integers followed by a Pascal string. A large variety of formatted data, and even a few varieties of unformatted data, can be stored in a resource.

The template resource is used to describe the exact structure of data stored in a resource. This structural definition contains descriptors for the types of data that can be stored in a resource of a particular type. These descriptors are commonly referred to as resource field types.

Resource field types are denoted by unique four byte values. Each different four byte value is a different format for data. The definitions of the resource field types to the data they describe can include specifiers for length, allowed and disallowed values, repeating values, counted repeating values, value ending indicators, and much, much more.

Like resource types, resource field types are often referred to by their ASCII equivalents. Though, do not mistake that resource field types are better referenced in 4D as longints.

Some commonly used resource field types are (without the apostrophes):

'DBYT'	1145198932	signed decimal byte
'DWRD'	1146573380	signed decimal word
'DLNG'	1145851463	signed decimal long
'REAL'	1380270412	single precision float
'DOUB'	1146049858	double precision float
'BOOL'	1112493900	boolean word
'CHAR'	1128808786	ASCII character
'PSTR'	1347638354	Pascal string
'CSTR'	1129534546	C string
'TXTS'	1415074899	sized text dump
'OCNT'	1329811028	one based count of list items
'LSTC'	1280529475	begin counted list item
'LSTE'	1280529477	end of any list item

By no means is the above list complete. Unfortunately, there is no organized, definitive registry of all of the resource field types. But, there are over a hundred defined resource field types that are almost universally accepted as standards. A list of many of these field types can be found at the 4D Zine web site (select the Freeware links, and do a search for "resource field types").

What is the structure of a template resource?

A template resource, as mentioned previously, has a well defined structure. This structure is a pair listing of resource field names and resource field types. The list can have as many items as are needed to fully describe the target resource type. The paired listings are of a single resource field type 'PSTR', denoting the name, followed by a single resource field type 'DLNG' , denoting the resource field type.

It would be best to illustrate this with a few examples.

The resource type 'TEXT' is a resource which most are already very familiar with. A resource of this type merely contains a sized block of text. The template resource for a resource of type 'TEXT' has the following appearance when opened with Resorcerer:

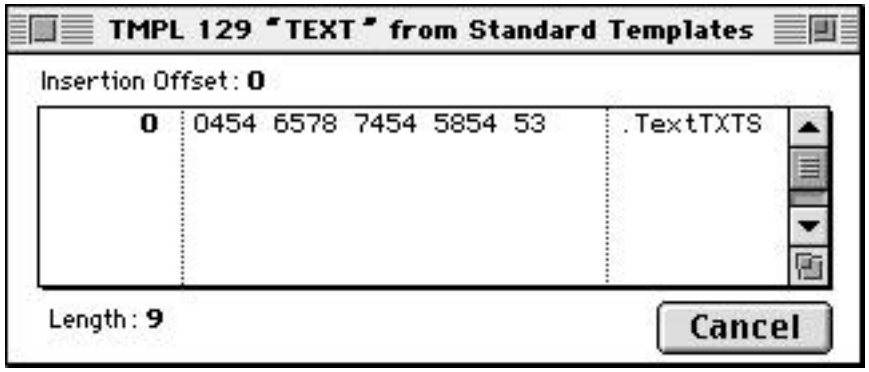

figure 2.7, the template resource for a resource of type 'TEXT' (viewed with the hex editor)

This template contains only a single pair of values. The first value is the field name, "Text", a 'PSTR'; the second value is a 'DLNG' denoting a resource field type of 'TXTS'.

By using this template resource, any resource of type 'TEXT' will contain a single value. This value will be a sized text block. Whenever an editor displays this resource for editing, the field will have a name of "Text".

A more complex example is that of a string list resource, 'STR#'. The template for this resource, when viewed in Resorcerer, has the following contents:

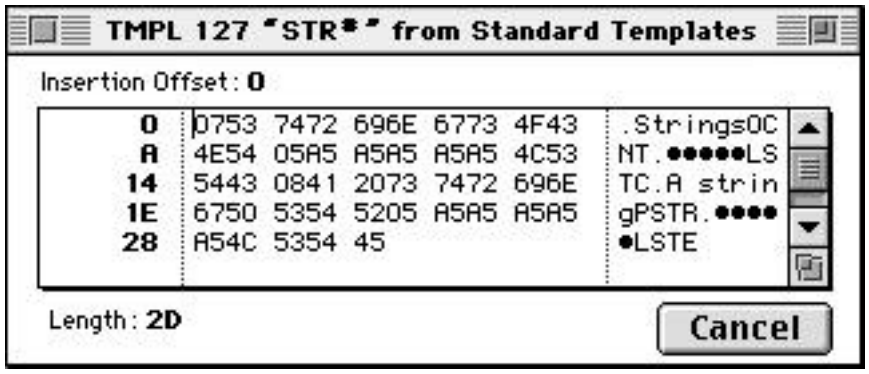

figure 2.8, the template resource for a resource of type 'STR#' (viewed with the hex editor)

As you probably already know, a string list resource contains a variable number of strings. The number of strings in a string list resource can vary; the number of strings in a string list resource can be different even within different resource IDs in the same document. This is variable structure is accomplished, and described, by the above template resource for a string list resource (figure 2.8).

By viewing the template resource for a string list resource, it can be seen what the structure of a string list resource contains. This consists of a two byte integer value, denoting the number of strings in the resource. This is followed by the strings, stored as Pascal strings.

The structure of a resource can be as complex as desired. Using the different resource field types available, any data structure can be stored in resources. This can include lists with multiple items, lists with items of different data types, lists within lists, and even binary fields containing unstructured data. There is really no limit to the

variability allowed within the structure of resources. The format of the template resources allows you to define whatever structure you so desire.

3. Resource Handling in the 4D World

As with any programming environment, 4th Dimension provides its own layer of abstraction for many areas of functionality. One of these areas of functionality that is abstracted is the way that resources are handled. To a large degree, 4D provides a fairly complete set of commands for handling resources.

Again, like other programming environments, understanding the lower layer is only "half the job". Understanding the basics of resources only allows you to understand what is being dealt with. As 4D programmers, you must also understand how 4D provides access to resources.

This task, as it encompasses resources, is complicated somewhat by the cross-platform nature of 4th Dimension. A basic understanding of the specifics of document forks as they relate to 4th Dimension helps in understanding the use of resources, especially under the cross-platform environment.

What does 4D Transporter do?

4D Transporter is known as the "magic tool" that makes your 4th Dimension database translate back and forth between the MacOS and Windows. It would seem that this tool would be very complex.

This is not so.

4D Transporter is actually a very simple application that deals with only one problem. This problem is the lack of document forks under Windows. As previously stated, the MacOS has an understanding of two different forks in documents: the data fork and the resource fork. But, under Windows, there is no equivalent document format. Under Windows, 4th Dimension deals with the different forks of a document by storing the forks in separate documents with different extensions.

4D Transporter merely takes the documents for a 4D project and provides the splitting (MacOS to Windows) or joining (Windows to MacOS) of the forks of a document, dependent upon the direction which the translation is going.

The exact actions taken by 4D Transporter, when moving a document from MacOS format to Windows, is as follows:

i) save the data fork of the MacOS document into a similarly named document with an appropriate extension;
ii) save the resource fork, if any, of the MacOS document into a similarly named document with an extension of '.RSR'.

There are specific rules for the naming conventions used for the new documents. And, there are specific document extensions under Windows for the different 4D related document types. Also, as an aside, the document creators and document types of the new documents are set to specific values so that the MacOS recognizes the documents as Windows documents.

When translating in the reverse direction, from Windows to MacOS, the same steps listed above are merely reversed.

There is nothing else of significance that 4D Transporter does. The "magic" of making documents work under both platforms is a result of the way in which the 4D product line has been developed. 4D Transporter actually plays no significant role in the cross-platform nature of the 4D product line. In actuality, it would be a fairly simple task to rewrite 4D Transporter *in* 4D.

How are resource files handled in 4D?

The first consideration in 4D when dealing with resources is specifying which resource file (under Windows) or document's resource fork (under MacOS) to work with.

There are two ways in which this can be done. First, you can use the resource file hierarchy (see the section *What is the resource hierarchy*, above). Second, you can use specify a specific resource document to use.

The two 4D commands **Open resource file** and **Create resource file** both return a resource file reference value. This value is the equivalent of a document reference for opened documents, but it is used for opened resource files. Every other resource handling command takes an optional resource file reference parameter. When this parameter is supplied with a resource command, then only the resource file referenced by the parameter is used for the resource operations. When this parameter is not supplied with a resource command, then the resource hierarchy is used to determine which, if any, resource file will be used for the resource operations.

There is only one exception to the optional resource file reference parameter; this is the **CLOSE RESOURCE FILE** command. When calling **CLOSE RESOURCE FILE**, the resource file reference parameter is mandatory (in fact, it is the only parameter).

Most single user 4D application are deployed using 4D Engine. But, when developing these applications, the 4th Dimension application is used to open and develop the database's structure file. This seemingly can present a problem when handling resources.

It may seem most advantageous to always use the resource file hierarchy when dealing with resources. Of course, this is the easiest, especially since 4D automatically opens the resource fork of the structure file when launched, thereby making it the first document in the resource file hierarchy. In many cases, this does work out to be the easiest way to deal with resource files.

But, as soon as any resource commands are used on other resource files or as soon as resources are referenced that do no exist in the structure file, there can be unexpected and possibly dangerous conditions created. Relying on the resource hierarchy does provide a level of convenience; it also makes your code reliant upon external factors which are beyond your control.

The safest and most foolproof method for handling resource files within 4D is to *always* use resource file references. This means that you should never rely on the resource file hierarchy. This prevents any possible unexpected behavior in your applications.

Resource file references are made available by merely using the **Open resource file** command. If a resource file is already open from use of this command, the same resource file reference will be returned in your code again. This is handy, especially for getting the resource file reference to the structure file. 4D automatically opens the resource fork of your structure file when launched; there is no harm at all in just calling **Open resource file** in your initialization methods to get the resource file reference for your structure.

It is a simple enough matter to write routines to get the path to the structure file for use in the **Open resource file**, whether this be under 4D, 4D Runtime, or merged with 4D Engine. This allows for a single routine to get and return the resource file reference for the structure file, under any conditions. Such a routine is provided in the sample code with this class (on the 4D Summit CD), and on the 4D Zine web site (search for "structure resource", under Freeware).

How are resource files handled under 4D Server?

Under 4D Server, the issue of resource files becomes a little more complex. There can be code that executes on the server and code that executes on the client machines.

On the client machine, 4D Client is the application which is actually running. Resources from the structure file are downloaded from the server to the client when the connection is first made. The resources are stored in the ".res" document. Figure 2.3, above, depicts the resource file hierarchy on a client machine under 4D Server.

To follow the above recommendations for always using the resource file reference parameter in resource command calls on client machines, the path to the ".res" document must be used. The path to the ".res" document will be different from machine to machine, and the form of the path is different from platform to platform. But, even with these variables, a simple, short routine utilizing standard 4D commands can be used effectively. A routine for doing this is provide din the sample code with this class (on the 4D Summit CD, and on the 4D Zine web site (search for "path", under Freeware).

When using resource commands within stored procedures, again the environment is a little different. The resource file hierarchy will contain the structure file of the database, followed by the 4D Server application. It is best to have a method which runs on the server, in the On Server Startup database method, that populates a variable with the resource file reference for the structure file of the database. This value can then be used by subsequent stored procedures when using resource commands.

What is resource substitution in 4D?

Resource substitution is a feature unique to the 4D development environment. It is a technique that simplifies the localization of an application. Basically, it is the substitution for the textual titles of certain with the contents of items in a string list resource.

There are a variety of 4D object titles that can reference directly items in a string list resource item for their names. This includes menu item names, menu names, and static text objects on forms. This allows for the contents of these objects, as they are displayed to the user, to be determined by the contents of a specific string list resource item.

This is done by using a particular format for the title of these items. This format is:

:{resource ID},{item number}

The leading colon and the separating comma are mandatory; the resource ID and item number are numeric values. This format for a title indicates to 4D to substitute the contents of the string list item referenced for the title of the object.

Another variation of this is to use interprocess variables in place of the actual numeric values for resource ID and item number. By setting the appropriate values in the interprocess variables, generic, reusable objects can have their titles changed programmatically; this is accomplished by merely referencing different string list resource IDs and item number in the interprocess variables.

4. A Detailed Example

Additions to the command language of version 6 of 4th Dimension gives complete access to resources. This means that any structure stored in the resource fork of a document can be used in support of your application design, development, and functionality.

Another interesting addition to the command language is the fairly complete set of commands for manipulating menus. This includes procedurally modifying menus, adding menu items, removing menu items, and controlling the state, title, style, and command-key equivalents.

There is already a standard structure for defining menus in resources. This is done through a series of related resources. Using this well-known structure, we can build a menu management system which is completely stored in resources.

A unique feature of 4D is that a particular method must be associated with each menu item. This can be simulated using a custom designed resource.

By following through these examples closely and studying the code that accompanies this class on the 4D Summit '98 CD, a very clear understanding of resources can be achieved in a fairly short period of time. It can give you the power to use resources to their full potential in your applications.

What resource types define menus?

The standard resource structure for menus covers many different resource types. The different resource types define individual menus, combine individual menus to form a menu bar, and apply color hierarchical support to menus.

The support that 4D natively has for menus is somewhat limited. This mainly encompasses the formation of menu bars with non-hierarchical menus within those menu bars. There is support for menu items styles, menu items marks, and command-key equivalents.

Though the structure of menus and menu bars can span many different resource types, limited the resource types to only those that define the features 4D supports yields a very simple resource structure. All of the features that 4D supports in menus is defined in only two standard resource type: 'MBAR' and 'MENU'. 'MBAR' resources combine different 'MENU' resources into a menu bar; 'MBAR' resources are used to define menus, including menu item titles, states, command-key equivalents, marks, and styles.

What is the structure of the 'MBAR' resource?

It is probably easiest to start by looking at the structure of the 'MBAR' resource. This structure is depicted in figure 4.1.

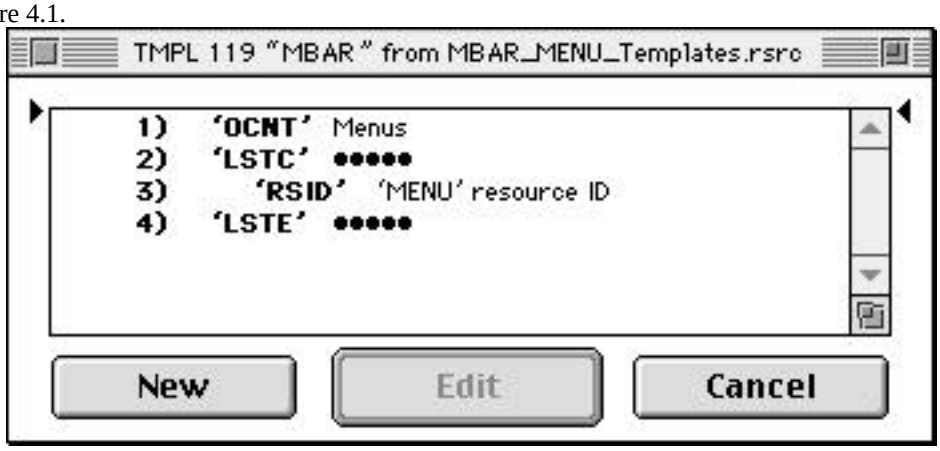

figure 4.1, the template resource for a resource of type 'MBAR' (viewed with the template editor)

It is easy to see that the structure of the 'MBAR' resource is merely that of a counted list of items, each item being the resource ID of a 'MENU' resource which is sequentially within the menu bar. The 'OCNT' template field type is a specific field type used as a counter for the number of items in a repeating list that follows. The 'LSTC' and 'LSTE' template field types indicate the beginning and ends, respectively, of the definition of the contents of the repeating list. The 'RSID' template field type merely indicates a numeric value that is a reference another resource, by ID.

Using standard 4D command language, it is fairly easy to read a resource of this type. The following method will read in a resource of type 'MBAR':

```
  `FUNCTION: RES_Load_MBAR
   `
  `Reads a resource of type 'MBAR'
   `
  `$0 is number of reference 'MENU' resource in specified 'MBAR' resource
  `   = -1 for specified 'MBAR resource not loaded
  `$1 is resource file reference
  `$2 is 'MBAR' resource ID
  `$3 is dereferenced array to put 'MENU' resource IDs into
   `
  `Deep Sky Technologies, Inc.
  `(561) 794-9494
  `http://www.deepskytech.com/
   `
C_BOOLEAN(◊qcSGW_v100)  `creator
  `================================================================
C_STRING(32;$s32_MethodName)
$s32_MethodName:="RES_Load_MBAR"
  `----------------------------------------------------------------------------
If (False)  `methods and variables used indirectly herein (for Insider compatibility)
    `indirect_variables
    `indirect_method_calls
End if   `false
  `================================================================
C_LONGINT($0;$iMenus)
$iMenus:=0  `default
C_TIME($1;$tResFileRef)
$tResFileRef:=$1
C_LONGINT($2;$iResourceID)
$iResourceID:=$2
C_POINTER($3;$pMenuResourceIDs)
$pMenuResourceIDs:=$3
  `----------------------------------------------------------------------------
C_STRING(4;$ks4_ResourceType)
$ks4_ResourceType:="MBAR"
  `----------------------------------------------------------------------------
C_TIME($nStructureResRef)
C_BLOB($zMBAR)
C_LONGINT($iBlobSize)
C_LONGINT($iOffset)
C_LONGINT($iIndex)
  `================================================================
  `check pointer for validity
If (Type($pMenuResourceIDs->)=LongInt array)
    `load the MBAR resource
  GET RESOURCE($ks4_ResourceType;$iResourceID;$zMBAR;$tResFileRef)
    `get the size of the resource
  $iBlobSize:=BLOB size($zMBAR)
    `make sure we got the resource
  If ($iBlobSize # 0)  `got it
      `start at the beginning of the blob
    $iOffset:=0
      `clear the dereferenced array
    ARR_Clear ($pMenuResourceIDs)
```

```
    `get the number of items in the resource
   $iMenus:=BLOB to integer($zMBAR;Native byte ordering;$iOffset)
    `make sure there are some menus in the menu bar
   If ($iMenus>0)  `there are some items
      `set up the array
     ARR_Add_Elements_to_End ($pMenuResourceIDs;$iMenus)
      `loop thru, getting all of the 'MENU' resource ids
     For ($iIndex;1;$iMenus;1)
      $pMenuResourceIDs->{$iIndex}:=BLOB to integer($zMBAR;Native byte ordering;$iOffset)
     End for
   End if
    `
  Else   `ERROR
   $iMenus:=-1
   ERROR_Handle (2900008;"";$s32_MethodName)
  End if
   `
Else   `ERROR
 $iMenus:=-1
 ERROR_Handle (2900006;"";$s32_MethodName)
End if
  `
$0:=$iMenus
  `eof
```

With this function, it is a simple matter to read in all of the 'MENU' resource IDs of a particular menu bar. Of course, without the ability to read in the menus, reading the menu bars would be useless...

What is the structure of the 'MENU' resource?

The template of the 'MENU' resource is shown in figure 4.2.

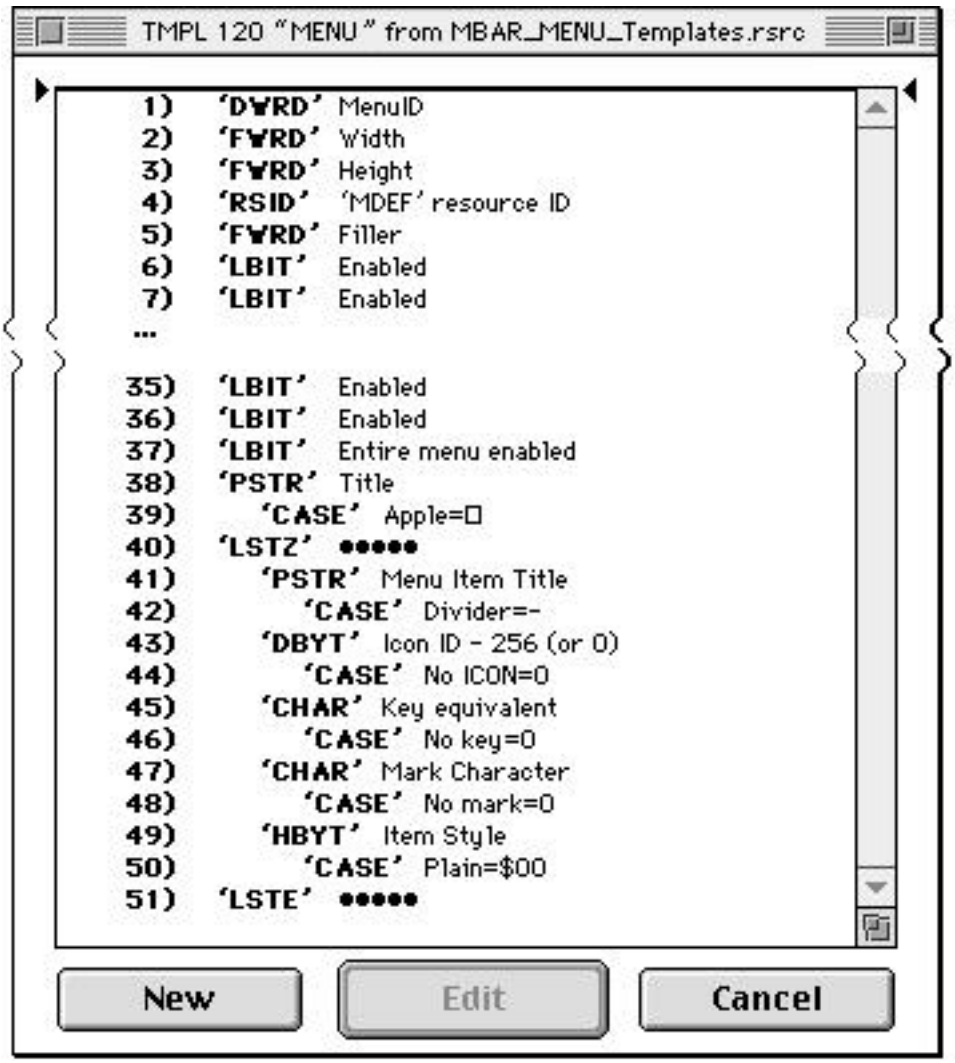

figure 4.2 the template resource for a resource of type 'MENU' (viewed with the template editor)

This is obviously a little more complicated than the previous template resources we have looked at. This resource template contain many more resource field types, and the list definition contains a much more involved repeating data structure. It is easiest to go from the top of the resource template definition and work our way down...

1) The Menu ID value is merely the resource ID of the current resource;
2, 3) 'FWRD' template field types indicate fillers for byte boundary alignments;
4) The 'MDEF' resource ID is used to handle references to code resources of type 'MDEF' for execution with items in this menu; for our purposes, is it not used and can be left as zero (0);
5) More filler;
6-36) The 'MENU' resource is limited by definition to handle a maximum of 31 menu item; these 31 bits are flags for the enabled state sequentially of the menu items in this resource;
37) The last bit, the 32nd, is a flag for the enabled state of the whole menu;
38, 39) A Pascal string of the actual menu title; a special case is given to indicate a menu which is to be used as the Apple menu; for our purposes in 4D, the special Apple menu case can be ignored;
40) A 'LSTZ' resource field type indicates the beginning of a repeating list which terminates with a zero byte;

The following items are repeated, as a whole data structure, once for each menu item in the 'MENU' resource:

41, 42) The title of each menu item, with a divider line be indicated by a hyphen ("-");
43, 44) Indicates the offset resource ID of the icon to display with this menu item; since this is a such a rarely used feature, we will just skip this piece of information;
45, 46) The command-key equivalent character, if any, for this menu item;
47, 48) The mark character, if any, for this menu item;
49, 50) The 'HBYT' resource field type is a single byte which is used to encode information; in this location, the first five bits of the byte are used as style flags for bold, italics, underlined, outlined, and shadowed, respectively;
51) The 'LSTE' resource field type indicates the end of the definition for the repeating list items; remember, this is a non-counted list terminating in a zeroed ($00) byte; this will be important for reading the resource in our code.

Armed with the above information, it is fairly easy to use native 4D code to write a routine for reading in and parsing a resource of type 'MENU'. The following routine accomplishes this quite quickly:

```
 `FUNCTION: RES_Load_MENU
  `
 `This loads a specified resource of type 'MENU'
  `
 `$0 is number of menu items loaded
 `   = -1 for error loading MENU resource
 `$1 is resource file reference
 `$2 is resource ID
 `$3 is dereferenced text array to hold menu item names
 `.   NOTE: index 0 is set to menu title
 `$4 is dereferenced boolean array to hold menu states
 `$5 is dereferenced longint array to hold menu styles
 `$6 is dereferenced longint array to hold menu keys
 `$7 is dereferenced longint array to hold mark characters
  `
 `Deep Sky Technologies, Inc.
 `(561) 794-9494
 `http://www.deepskytech.com/
  `
C_BOOLEAN(◊qcSGW_v100)  `creator
  `=================================================================
C_STRING(32;$s32_MethodName)
$s32_MethodName:="RES_Load_MENU"
  `-----------------------------------------------------------------------
If (False)  `methods and variables used indirectly herein (for Insider compatibility)
    `indirect_variables
    `indirect_method_calls
End if   `false
  `=================================================================
C_LONGINT($0)
$0:=0  `default
C_TIME($1;$tResFileRef)
$tResFileRef:=$1
C_LONGINT($2;$iResourceID)
$iResourceID:=$2
C_POINTER($3;$pMenuItems)
```

```
$pMenuItems:=$3
C_POINTER($4;$pMenuStates)
$pMenuStates:=$4
C_POINTER($5;$pMenuStyles)
$pMenuStyles:=$5
C_POINTER($6;$pMenuKeys)
$pMenuKeys:=$6
C_POINTER($7;$pMenuMarks)
$pMenuMarks:=$7
  `------------------------------------------------------------------------------
C_STRING(4;$ks4_ResourceType)
$ks4_ResourceType:="MENU"
  `------------------------------------------------------------------------------
C_BOOLEAN($qErrorOccurred)
$qErrorOccurred:=False  `default, used for flow control of code
C_BLOB($zTheResource)  `the 'MENU' resource read into RAM
C_LONGINT($iOffset)  `byte offset within 'MENU' resource
C_LONGINT($iResourceSize)  `total number of bytes in 'MENU' resource
C_LONGINT($iMenu)  `current menu number being dealt with
C_LONGINT($iMenuItem)  `current menu item being dealt with
C_LONGINT($iMenuStates)
C_TEXT($xMenuItem)
  `==================================================================
  `check pointers for validity
If (Not((Type($pMenuItems->)=Text array) | (Type($pMenuItems->)=Array 2D)))
  $qErrorOccurred:=True
  ERROR_Handle (2900004;"";$s32_MethodName)
End if
If (Not((Type($pMenuStates->)=Boolean array) | (Type($pMenuStates->)=Array 2D)))
  $qErrorOccurred:=True
  ERROR_Handle (2900005;"";$s32_MethodName)
End if
If (Not((Type($pMenuStyles->)=LongInt array) | (Type($pMenuStyles->)=Array 2D)))
  $qErrorOccurred:=True
  ERROR_Handle (2900006;"";$s32_MethodName)
End if
If (Not((Type($pMenuKeys->)=LongInt array) | (Type($pMenuKeys->)=Array 2D)))
  $qErrorOccurred:=True
  ERROR_Handle (2900006;"";$s32_MethodName)
End if
If (Not((Type($pMenuMarks->)=LongInt array) | (Type($pMenuMarks->)=Array 2D)))
  $qErrorOccurred:=True
  ERROR_Handle (2900006;"";$s32_MethodName)
End if
  `
If (Not($qErrorOccurred))
    `clear the arrays
  ARR_Clear ($pMenuItems)
  ARR_Clear ($pMenuStates)
  ARR_Clear ($pMenuStyles)
  ARR_Clear ($pMenuKeys)
  ARR_Clear ($pMenuMarks)
    `load the resource
  GET RESOURCE($ks4_ResourceType;$iResourceID;$zTheResource;$tResFileRef)
```

```
 If (OK=1)  `found and read it
    `
  $iResourceSize:=BLOB size($zTheResource)-1
  $iOffset:=10  `skip early junk
   `extract all of the states
  $iMenuStates:=BLOB to longint($zTheResource;Native byte ordering;$iOffset)
    `   NOTE: macintosh byte ordering fails on a WIntel
   `extract the menu title
  $xMenuTitle:=BLOB to text($zTheResource;Pascal string;$iOffset;0)
    `loop through menu items
  $iMenuItem:=0
  While ($iOffset<$iResourceSize)
     `
    $iMenuItem:=$iMenuItem+1  `increment for menu item number
     `
    INSERT ELEMENT($pMenuItems->;Size of array($pMenuItems->)+1;1)
    INSERT ELEMENT($pMenuStates->;Size of array($pMenuStates->)+1;1)
    INSERT ELEMENT($pMenuStyles->;Size of array($pMenuStyles->)+1;1)
    INSERT ELEMENT($pMenuKeys->;Size of array($pMenuKeys->)+1;1)
    INSERT ELEMENT($pMenuMarks->;Size of array($pMenuMarks->)+1;1)
     `
      `extract the menu item
    $xMenuItem:=BLOB to text($zTheResource;Pascal string;$iOffset;0)
    $pMenuItems->{$iMenuItem}:=$xMenuItem
      `skip icon ID
    $iOffset:=$iOffset+1
      `extract the command key equivalent
    $pMenuKeys->{$iMenuItem}:=Ascii(BLOB to text($zTheResource;Text without length;$iOffset;1))

      `extract the mark character
    $pMenuMarks->{$iMenuItem}:=Ascii(BLOB to text($zTheResource;Text without length;$iOffset;1))

      `extract the style
    $pMenuStyles->{$iMenuItem}:=Ascii(BLOB to text($zTheResource;Text without length;$iOffset;1))

      `store menu state
    $pMenuStates->{$iMenuItem}:=($iMenuStates ?? $iMenuItem)
     `
  End while
    `
  $pMenuItems->{0}:=$xMenuTitle
    `
  $0:=$iMenuItem
    `
 Else   `failed to read it
  $0:=-1
   ERROR_Handle (2900008;"Resource ID "+String($iResourceID);$s32_MethodName)
    `
 End if
   `
   `
Else   `ERROR occurred in parameter checking
 $0:=-1
End if
```

```
`eof
```

What else is needed?

Using the commands available in 4D, menu bar information can be extracted from 'MBAR' resources. The specified 'MENU' resources for a menu bar can then be loaded. And, with the information supplied in the menu bars, menus can be completely generated within 4D. This allows for a complete system of menu management using our new knowledge of resources.

But, there is one piece of information missing. The menu bars and menus can be built completely from the resources. But, once built, how will by made to function?

Using the **On Menu Selected** form event can let us trap events in the menus; we can even trap which menu and menu items was selected. With this information, we can programmatically respond to the selection. But, with this approach, the modularity of constructing the menus completely in resources is lost. A different, more flexible, approach is needed.

As often happens when in this situation, the natural command to consider is **EXECUTE**. If in some way we could directly associate menu items with method names, we could build a system to respond to menu item selections. Of course, we want to use resources for this, making it easier to modify the information in the future, if needed.

What is the 'MENV' resource?

The 'MENV' resource is what will allow us to associate specific menu items with method names. The 'MENV' resource is a resource type which we will construct specifically for the purpose of associating these pairs of information together.

NOTE: using all uppercase ASCII values in the name of a resource which we create actually violates the guidelines set down by Apple. Apple reserves all resource types consisting of all uppercase or all lowercase ASCII values. I won't tell if you won't, though...

As stated previously, there are plenty of resource types already available. But, there is nothing wrong with creating your own resource types for specific purposes in your applications. This is a case in which a dedicated resource type would probably be useful for our purposes. Hence, we shall create this resource for this application.

We know that we need a very simple listing of paired values. Since this is involve some form of arbitrary list length, we might as well make the list a counted list. The following resource template will work just fine for our needs (figure 4.3):

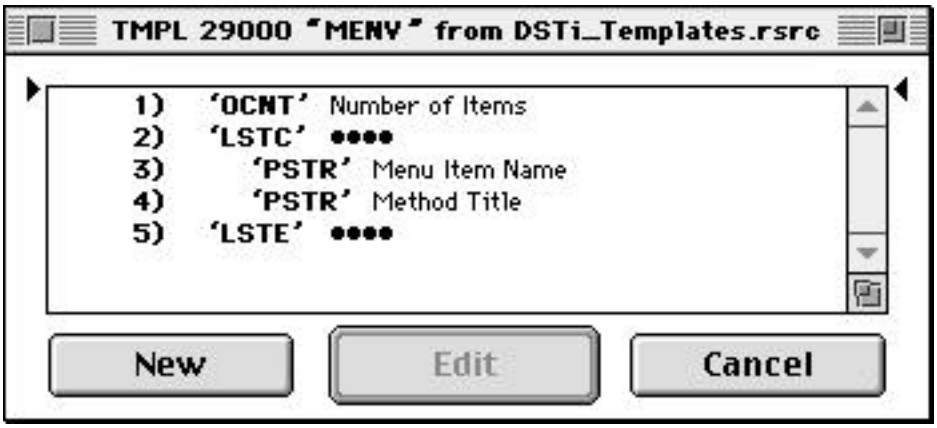

figure 4.3 the template resource for a resource of type 'MENV' (viewed with the template editor)

This resource template provides for a counted list of items, of arbitrary length, with each list item consisting of Pascal strings for the menu item name and method name.

The code to read in a resource of type 'MENV' is then quite simple.

```
  `FUNCTION: RES_Load_MENV
   `
  `Loads a resource of type 'MENV'
   `
  `$0 is number of menu items in the 'MENV' resource
  `   = -1 for an error has occurred
  `$1 is resource file reference
  `$2 is reosurce ID of 'MENV' resource to load
  `$3 is dereferenced array to hold menu item names
  `$4 is dereferenced array to hold method names
   `
  `Deep Sky Technologies, Inc.
  `(561) 794-9494
  `http://www.deepskytech.com/
   `
C_BOOLEAN(◊qcSGW_v100)  `creator
  `===========================================================================
C_STRING(32;$s32_MethodName)
$s32_MethodName:="RES_Load_MENV"
  `---------------------------------------------------------------------------
If (False)  `methods and variables used indirectly herein (for Insider compatibility)
    `indirect_variables
    `indirect_method_calls
End if   `false
  `===========================================================================
C_LONGINT($0;$iMenuItems)
$iMenuItems:=0  `default
C_TIME($1;$tResFileRef)
$tResFileRef:=$1
C_LONGINT($2;$iResourceID)
$iResourceID:=$2
C_POINTER($3;$pMenuItemNames)
$pMenuItemNames:=$3
C_POINTER($4;$pMethodNames)
$pMethodNames:=$4
  `---------------------------------------------------------------------------
C_STRING(4;$ks4_ResourceType)
$ks4_ResourceType:="MENV"
  `---------------------------------------------------------------------------
C_BOOLEAN($qErrorOccurred)
$qErrorOccurred:=False  `default
C_TIME($nStructureResRef)
C_BLOB($zMenuItems)
C_LONGINT($iBlobSize)
C_LONGINT($iOffset)
C_LONGINT($iIndex)
```

```
  `================================================================
  `check pointers for validity
If (Not((Type($pMenuItemNames->)=Text array) | (Type($pMenuItemNames->)=String array)))
  $qErrorOccurred:=True
  ERROR_Handle (2900004;"";$s32_MethodName)
End if
If (Not((Type($pMethodNames->)=Text array) | (Type($pMethodNames->)=String array)))
  $qErrorOccurred:=True
  ERROR_Handle (2900004;"";$s32_MethodName)
End if
   `
  `make sure there have been no errors so far
If (Not($qErrorOccurred))  `no error yet
    `load the MENV resource
  GET RESOURCE($ks4_ResourceType;$iResourceID;$zMenuItems;$tResFileRef)
    `get the size of the resource
  $iBlobSize:=BLOB size($zMenuItems)
    `make sure we got the resource
  If ($iBlobSize # 0)  `got it
      `make sure there is some data in the resource
    If ($iBlobSize>1)  `there is some data
        `start at the beginning
      $iOffset:=0
        `get the number of items
      $iMenuItems:=BLOB to integer($zMenuItems;Macintosh byte ordering;$iOffset)
        `   NOTE: native byte ordering fails on a WIntel
        `clear the arrays
      ARR_Clear ($pMenuItemNames)
      ARR_Clear ($pMethodNames)
        `make sure there are some menu items in the resource
      If ($iMenuItems # 0)  `there are some menu items
          `set the siz eof the arrays
        ARR_Add_Elements_to_End ($pMenuItemNames;$iMenuItems)
        ARR_Add_Elements_to_End ($pMethodNames;$iMenuItems)
          `loop, getting all of the menu items
        For ($iIndex;1;$iMenuItems;1)
          $pMenuItemNames->{$iIndex}:=BLOB to text($zMenuItems;Pascal string;$iOffset)
          $pMethodNames->{$iIndex}:=BLOB to text($zMenuItems;Pascal string;$iOffset)
       End for
      Else   `there are no items to extract from the resource
       $iMenuItems:=0
     End if
       `
    Else   `there are no items to extract from the resource
     $iMenuItems:=-1
     ERROR_Handle (2900008;"";$s32_MethodName)
    End if
      `
  Else   `ERROR
   $iMenuItems:=-1
   ERROR_Handle (2900008;"";$s32MethodName)
  End if
    `
Else   `an error has occurred
```

```
  $iMenuItems:=-1
End if
   `
$0:=$iMenuItems
   `eof
```

What about all of the menu generation code?

The remainder of this menu management system involves code which will take the data of the about routines and handle the creation, modification, and storage of the menu bars and menus in multiple processes. All of this code is beyond the scope of this class, as none of it directly involves resources.

The sample database that is on the 4D Summit '98 CD (and also available on the 4D Zine web site) has all of the code for using these routines. It contains a very simple, yet powerful, window, process, and menu management system based on these routines. It even handles a dynamic Windows menu for all of the open windows in the application.

Glossary

data fork: under the MacOS, the data fork is one of the system defined content areas of a document; it is an unstructured content area for storage of code and data; under Windows, there is no system level equivalent to a data fork.

FILO stack: a stack of data organized on a first in, last out basis for pushing and popping.

resource: a structured data object stored in the resource fork of a document (under MacOS) or in a resource file (under Windows).

resource field type: an atomic data, branch, case, or list definition within resource templates.

resource file chain: same as *resource hierarchy*.

resource file reference: within 4D, a unique identifier for an open resource fork or resource file.

resource fork: under the MacOS, the resource fork is one of the system defined content areas of a document; it is a structured content area for storage of code and data; under Windows, there is no system level equivalent to a resource fork.

resource hierarchy: a FILO stack of resource forks which are available within the current context.

resource ID: a unique, signed 2 byte number identifying a particular resource within one or more resources of a particular resource type.

resource name: a non-unique Pascal string identifier of a resource.

resource number: same as *resource ID*.

resource properties: a series of flags with predefined purposes that are associated with every resource.

resource substitution: in 4D, the technique of reference items in a string resource (type 'STR#') as replacements for textual items.

resource template: a predefined resource type used to define the structure of other resource types.

resource type: a 4 byte identifying value associated with each resource; resources of the same resource type share the same structure.

windows extension: the three character extension given to all documents stored under Windows (i.e. "8 dot 3").