THE TEMPLATE EDITOR

INTRODUCTION

This chapter explains how to use Resorcerer's Template Editor to design descriptions, called *templates*, of your own custom resources. The template-driven Data Editor then lets you edit your custom types using the descriptions you've built with the Template Editor. The Data Editor is more fully explained in the "Data Editor" chapter.

You should be familiar with the basics of editing resources, as explained in the "Editing Resources" chapter earlier in the manual. Also, the Template Editor has an interface that is nearly identical to that of the String List Editor, so you may want to familiarize yourself with its general operation (such as cutting and pasting fields/strings, printing fields/strings, etc.) in the chapter, "The String List Editor".

TOPICS COVERED

- Custom resource types
- Creating a resource template
- Using the Template Editor
- Template field types
- Examples of templates
- Designing filters

CUSTOM RESOURCE TYPES

Only a few hundred resource types, such as 'DLOG's, 'ICON's, 'STR#'s, have been publicly defined by Apple and its developer community. The description of the internal structures of these resources is documented in various programming manuals (primarily Apple's own *Inside Macintosh*) and Rez template files. Resorcerer has many Editors dedicated to making it easy for you to edit the most common of these standard resources.

Although these predefined types are often sufficient for many Mac applications, you may want to have your own custom resource types that are private to your application. To create a custom resource type you will first want to find a new four-character name for your type that doesn't conflict with existing resource types; your program can then use the Resource Manager to deliver that resource to it at the appropriate times.

The Resource Manager in the Macintosh is designed to access resources whose types are represented by any combination of four ASCII characters. Thus, there are as many possible types as there are four 8-bit character combinations, which is to say, something like 4 billion types available.

Note: Apple has reserved resource type names consisting of all lowercase characters for its own use. Third-party applications should not use all lowercase type names for any custom resources.

For example, Resorcerer is a normal Mac application that uses both standard and custom resources. One type of custom resource it uses is its 'RSYN' resource. 'RSYN' is not among the standard Mac types; the name was chosen by the program's author to refer to Resorcerer's resource synonym table, which is kept as an 'RSYN' resource in Resorcerer's own resources (for more on what it's for, see the Synonym Preferences section of the "Preferences" chapter).

RESOURCE TEMPLATES

Resorcerer can't anticipate all the possible custom resource types that every Mac programmer might want to create and edit. With upwards of 4 billion possible custom types, there can never be more than a relatively few resource Editors in Resorcerer (or any other general resource editing program) dedicated to helping you edit the standard documented Mac resources.

Although any resource, regardless of type, can always be edited using the general-purpose Hex Editor, this is nearly always a tedious, error-prone, and unpleasant task. The whole purpose of the Hex Editor is to hide the structure of the data from you, and it is the editing method of last resort.

A better solution is to describe to Resorcerer the structure of your custom resource. Based on a description of that type of resource, Resorcerer can then customize its Data Editor to let you edit resources of your described custom type.

Resorcerer keeps the descriptions of the structure of custom resources in what are called *resource templates*. Each template is kept as a resource of type 'TMPL' whose resource name begins with the same four characters as the name of the custom resource type it describes. Each template is essentially a list of the data fields (their types and descriptions) that, in sequential order, describe the structure of the packed data in your custom resource.

Note: With one minor exception, Resorcerer's 'TMPL' resources are upwardly compatible with 'TMPL' resources found in Apple's old ResEdit program. However, Resorcerer supports more than three times as many field types as ResEdit. These field types are documented in the "Template field types" section later in this chapter.

FIXED-LENGTH AND VARIABLE-LENGTH DATA

When each part of a resource data structure has a known fixed length, it is very easy to compute the offset into the resource data at which to access that part of the resource. However if any part of a resource has a variable size – for example, an embedded, unpadded string – then it is no longer so easy to compute the offset of the start of any field that occurs later in the data. This means that algorithms that access the resource must at some point begin scanning the data sequentially to find a field of interest if it occurs after a variable-length item in the data. For large resources, this scanning can become slow.

One way to speed up the scanning of resource data is to include explicit length fields in the data. These fields precede the variable-length item so that your application's (or the Toolbox's) scanning algorithm can compute where the end of the item is without scanning through its data. For very complicated sequences of variable-length items the Data Editor

supports explicit *sizeof* and *skip offsets* fields that let you embed automatically computed length information for anything you want.

For example, variable-length Pascal strings begin with a single length byte that can be used to find the start of the first field after the string; C strings, on the other hand, end with a null byte so that they must be scanned in order to find their ends (in C, this is the price of being able to have arbitrarily long strings). Lists embedded in resource data are another example of a higher level variable-length object that must have an initial count field to indicate how long the list is.

Placing size information about internal structures in a resource is highly recommended (not to mention absolutely necessary!) if you want to design a resource to be both extensible and backwardly compatible.

FILTERED TEMPLATES

Templates are designed to support the description of scannable sequences of individual data fields, both fixed- and variable-length. In order to access variable-length fields randomly (that is, without scanning), it is necessary to design an *index* at the start of the resource data. The index is basically a table of contents that consists of a list of fixed length fields containing (at the very least) offsets to the starts of the later variable-length fields. More complicated indexes might also contain explicit length fields, key fields, or other fixed-length data. There are many different ways to do this, but unfortunately there is no easy way to design such a data structure directly using the "language" of 'TMPL' field types.

The Data Editor supports *filtered templates* to solve the above problem in a general way. A filtered template for a particular resource type describes a closely related, editable data structure that you can edit, but which is not the same as the resource data. When the Data Editor recognizes that the template is a filtered template, the Editor will automatically pre- and post-process the actual resource data into and out of the form described by the filtered template. This conversion is performed by a *filter* — that is, a code resource of type 'FLTR' that accompanies the 'TMPL' resource.

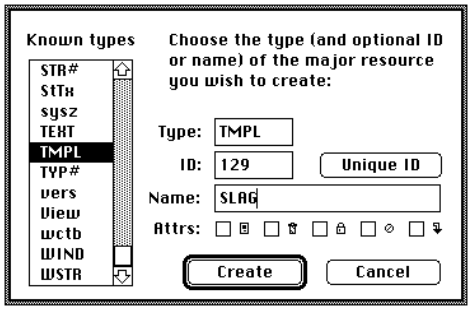
You can design the structural rules of a given resource data type into its filter, which converts the data into a scannable and editable form. After editing the converted data, the filter scans the data and re-inserts whatever structural information it needs–such as an index–back into the resource. With filters, you can bring the full power of any compiled

high-level language to bear on the maintenance of complex resource data structures, without at the same time losing the power of interactively editing the parts of the resource data that you are more likely to care about. Structural information such as indexes, skip offsets, and the like are, after all, for the benefit of the scanning algorithms, not for the benefit of the user who would just as soon ignore the structural information altogether while editing.

For more on filtered templates, see the description of the 'FLTR' field type and the "Designing Filters" section later in this chapter.

CREATING A RESOURCE TEMPLATE

Creating a new 'TMPL' resource in a file you've opened is no different from creating any other new resource (see the "Editing Resources" chapter). Make sure the File Window is in front, click once in its Types List (preferably on the **TMPL** entry if there is one), and then click the **New** button.



Resorcerer will ask you to specify the various attributes of your new 'TMPL' resource. The resource ID and attributes can be whatever you want; however, the first four characters of the name of the 'TMPL' resource must be the four-character resource type that the 'TMPL' is going to describe.

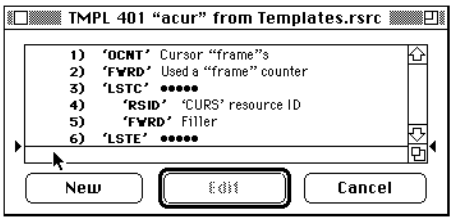
Note: For future compatibilty, we ask that you avoid using four space characters (' ') as a resource type.

For example, if the template is going to describe your custom resource of type 'SLAG', you should type the word "SLAG" into the name field of the dialog. Click in the **Create** button to create the new 'TMPL'. Resorcerer asks the Template Editor to create a new empty template, which it displays for you in an editing dialog window.

Note: 'TMPL' resources have a structure similar to standard 'STR#' (string list) resources, and in fact the same Editor edits both. The list-based interface is almost identical.

USING THE TEMPLATE EDITOR

Every template ('TMPL') resource is nothing but a list of *fields*, where each field consists of two parts: a four-character code, called the field *type*, and a Pascal string, called the field *label*. The label is a string that usually contains a readable description of the field's purpose. The Data Editor will use the label to identify the field for you.



When the Template Editor opens a template, it creates a standard list with one field per entry, arranged vertically. Each list entry displays, from left to right, the index of the field into the resource (counting from 1); the field type in boldface and surrounded by single quotes; and finally the value of the field's label string. Fields are indented to show the various nesting levels.

The small triangular handles on either side of the list area in the editing dialog show you the position of the list insertion caret. If there is no currently selected field in the list, the caret will be blinking a horizontal line at the position in the list where any insertion or pasting of fields will occur. If there is at least one field selected, the horizontal line will not be blinking; the caret position, however, remains visible via the triangular handles on either side of the list.

To position the list insertion caret, click on either triangular handle and drag it up or down until the horizontal caret is between the two list entries where you want to insert a new entry. If you drag the handle above or below the top or bottom of the list, the field list will automatically scroll.

A grow box in the lower right corner of the list lets you grow the list to any convenient size (the dialog window that contains the list will grow to accommodate the new list size), and the ZoomBox in the window's upper right corner grows or shrinks the window to the size of the screen on which it is found.

Below the list are three buttons: **New**, **Edit**, and **Cancel**.

The **New** button creates a new field, inserts it between the two list entries that are separated by the list insertion caret, and opens the new field for editing. You can also double-click on a list insertion caret handle to create a

new field at the position of the caret.

The **Edit** button opens an editing dialog window for each selected field in the template.

The **Cancel** button throws away any changes you've made to the template resource since you opened it, and closes its editing dialog window.

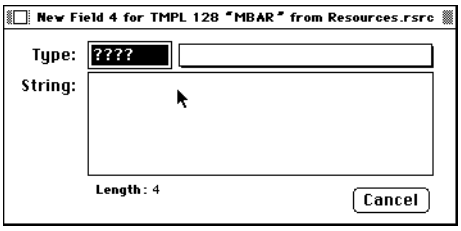
You can select any possible set of fields from the template field list. You can cut, copy, or clear the current selection when you choose **Cut**, **Copy**, or **Clear** from the **Edit** menu (tapping the Delete key also Clears the current selection); the fields in the current selection can be rearranged without cutting and pasting when you choose **Reorder Fields** from the **Template** menu; and you can open all selected fields for editing by clicking on the **Edit** button.

To select all fields within a given indentation level, use the **Select All** command in the **Edit** menu. If all selected fields are already at the same indentation level, then the command will select all fields at the previous indentation level. This lets you use **Select All** to check for balanced fields that begin and end indentation levels.

Sorcery: ⌘ . (period) is the keyboard equivalent of the **Cancel** button.
⌘A is the keyboard equivalent of the **Select All** command.
The Shift key extends the current selection of the list.
The ⌘ key toggles the selection status of any field you click on.

ADDING A NEW TEMPLATE OR RESOURCE FIELD

Click in the **New** button to add a new field wherever the horizontal list caret is positioned. The Editor opens an editing dialog for the new field. The field type is initially set to '????' with an empty label string.



Sorcery: ⌘N is the keyboard equivalent of the **New** button.
Double-clicking on a triangular handle on either side of the list insertion caret creates a new field at that position.
You can hold the Option key down while creating new fields to suppress opening their editing dialog windows.

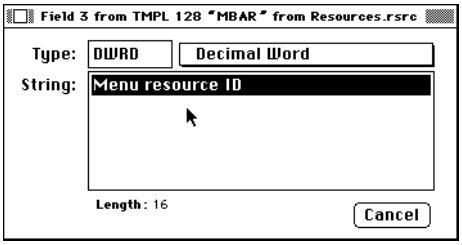
If you have existing custom resources that have been created with a template, your addition of a new data field type to the template renders it incapable of accurately opening the old resources, whose data has not yet changed to the new format.

Resorcerer supports special insertion and deletion field types that can alleviate the above problem. The insertion fields let you to insert data at any position in an existing resource when you open it, leaving the data there when you close the resource. The deletion fields let you delete existing data at any position when you close the resource.

Each of these field types begins with a '+' for insertion, or a '-' for deletion. For more on their operation, see the next section, "Template Field Types", and the tutorial section at the end of the chapter.

EDITING AN EXISTING TEMPLATE FIELD

Once a field has been defined in a 'TMPL' resource, you can select it by clicking on it and then edit it by clicking in the **Edit** button, or by double-clicking directly on the field. The Editor opens a field editing dialog window to let you change either the field type or its label string.



The field editing dialog displays a small editing text box in which you should enter a four-character field type (described below); a larger editing box in which you should enter a descriptive label string; and a popup menu next to the field type that lets you choose from all the legal four-character field types that Resorcerer's Data Editor currently supports. Since there are some 120 field types, the popup menu lets you see and choose from the various types so you don't have to remember them all. Each entry in the menu gives a written description of the field type, followed by the four-character type code. Related entries in the menu are grouped together and separated by divider lines.

The field label string identifies the field when the Data Editor uses the template to present your resource's field values to you. The Data Editor allows plenty of room for your label strings. Usually, there is no need to abbreviate words, and full readable descriptions are encouraged.

When you have finished entering the field type and its label, click in the GoAway box to install the changed or new field.

Sorcery: ⌘W is the key equivalent of closing and saving the field. ⌘ . (period) is the key equivalent of the **Cancel** button.

IMPORTING AND EXPORTING TEMPLATE FIELDS AS TEXT

Once you've mastered the language of template fields, you can design your templates as text using your development system's source code text editor. For large and complicated data structures this will be a much faster method of resource design. In addition, you can use the tools of your text editor to manipulate the declarations however you want. For example, you might want to convert a set of C `#define`'s or `enum`'s into a set of template 'CASE' fields.

The Template Editor will cut or copy any selection of template fields. You can paste these into your text editor and work on them there. Then when you're done, select the fields in your text editor, copy them to the clipboard, switch back to Resorcerer, and paste them into your template at the position of the list insertion caret.

The Template Editor scans the text, one field per line. It ignores all indentation (spaces and tabs) at the start of each line, so you can indent in your text editor. The next four characters, including spaces, are taken as the field type. Following this type there must be exactly one character, usually a space or a tab, which is ignored. After this, all characters to the end of the line are converted to a Pascal string and installed as the field's label. Blank lines are ignored.

PRINTING A TEMPLATE

You can print the state of your template at any time. To do so, choose **Print...to Printer** from the **File** menu. The Editor prints an indented listing of the template fields in the current text style.

TEMPLATE FIELD TYPES

Each field in a template consists of a descriptive label string and a four-character field type. Nearly all field types declare how the byte(s) at a particular position in your custom resource should be interpreted. The position of the data of any particular field is implicitly defined by the sequence of fields that occur prior to it in the resource.

Resorcerer supports about 120 template data field types. Types that are not compatible with Apple's ResEdit 2.1, or which can be used in circumstances that are legal in Resorcerer but not in ResEdit, are marked with a • .

Integer Numeric Values

DBYT		Decimal byte
DWRD		Decimal word
DLNG		Decimal long
UBYT	•	Unsigned (decimal) byte
UWRD	•	Unsigned (decimal) word
ULNG	•	Unsigned (decimal) long
HBYT		Hex byte
HWRD		Hex word
HLNG		Hex long

Bit and Bit Field Values

BBIT		Byte bit (for bits labeled 7 to 0)
BB*nn*	•	Unsigned decimal byte bit field
WBIT	•	Word bit (for bits labeled 15 to 0)
WB*nn*	•	Unsigned decimal word bit field
LBIT	•	Long bit (for bits labeled 31 to 0)
LB*nn*	•	Unsigned decimal long bit field
BFLG	•	Byte boolean flag (low-order bit 0 only)
WFLG	•	Word boolean flag (low-order bit 0 only)
LFLG	•	Long boolean flag (low-order bit 0 only)

Floating and Fixed Point Values

REAL	•	Single 32-bit floating point number
DOUB	•	Double 64-bit floating point number
EXTN	•	Extended 80-bit floating point number
XT96	•	Extended 96-bit floating point number
UNIV	•	THINK C Universal 96-bit floating point

FIXD	•	32-bit `Fixed` point number (16:16)
FRAC	•	32-bit `Fract` number (2:30)
SFRC	•	16-bit `SmallFract` number (0:16)
FWID	•	16-bit font width number (4:12)

Miscellaneous Types

RSID	•	Resource ID (signed decimal word) reference
BOOL		Boolean word (user-configuarable True or False)
CHAR		Single byte character
TNAM		Four-character OS or resource type name
DATE	•	Long system date/time
MDAT	•	Automatically set modification date
PNT	•	QuickDraw Point
RECT		QuickDraw Rectangle
COLR	•	A Color QuickDraw RGB Triplet
CODE	•	Code dump for disassembling rest of resource

Character String Values

PSTR		Pascal string
ESTR		Pascal string, even-padded
PPST	•	Pascal string, even-padded, pad-included
OSTR		Pascal string, odd-padded
CSTR		C string
ECST		C string, even-padded
OCST		C string, odd-padded
BSTR	•	Byte length-encoded string (same as PSTR)
WSTR		Word length-encoded string
LSTR		Long length-encoded string
TXTS	•	Sized text dump
P*nmm*	•	Pascal string in fixed-length field of $*nmm* bytes
C*nmm*		C string in fixed-length field of $*nmm* bytes
T*nmm*	•	Text in fixed-length field of $*nmm* bytes

Labeling

DVDR	•	Divider line and/or section label

Lists of Repeated Items

OCNT	•	Word containing number of subsequent list items
BCNT	•	Byte containing number of subsequent list items
LCNT	•	Long containing number of subsequent list items
ZCNT	•	Word containing 0-based count of list items

LZCT	•	Long containing 0-based count of list items
FCNT	•	Fixed count array, count in label string
LSTC		Start of counted list item
LSTB		Begin non-counted list item
LSTS	•	Start of sized list
LSTZ		Begin list of items ending in zero byte
LSTE		List item end
SELF	•	Entire item is recursive instance of this template

Alignment

AWRD		Align next field on word boundary (uneditable)
ALNG		Align next field on long boundary (uneditable)
AL08	•	Align next field on 8-byte boundary (uneditable)
AL16	•	Align next field on 16-byte boundary (uneditable)

Automatically Set Skip Offset and Sizeof Values

BSKP	•	Offset to SKPE, stored in a byte
SKIP	•	Offset to SKPE, stored in a word
LSKP	•	Offset to SKPE, stored in a long
BSIZ	•	Size of following data, stored in a byte
WSIZ	•	Size of following data, stored in a word
LSIZ	•	Size of following data, stored in a long
SKPE	•	End of skip or sizeof

Key Values for Variant Items

KBYT	•	Signed decimal byte key
KWRD	•	Signed decimal word key
KLNG	•	Signed decimal long key
KUBT	•	Unsigned decimal byte key
KUWD	•	Unsigned decimal word key
KULG	•	Unsigned decimal long key
KHBT	•	Hex byte key
KHWD	•	Hex word key
KHLG	•	Hex long key
KCHR	•	Single character key
KTYP	•	4-character type key
KRID	•	Key off of resource ID, not data

Symbolic Constant Definition

CASE	•	Symbolic or default value for previous data field

Keyed Items

KEYB	•	Begin keyed item for associated key CASE
KEYE	•	End of keyed item

Data Filter for Pre- and Post-Processing

FLTR	•	Pre- and post-process data with compiled filter

Hex Data (Unknown Format)

BHEX	•	Byte byte count followed by that many bytes
WHEX	•	Word byte count followed by that many bytes
LHEX	•	Long byte count followed by that many bytes
BSHX	•	Byte byte count followed by that many bytes - 1
WSHX	•	Word byte count followed by that many bytes - 2
LSHX	•	Long byte count followed by that many bytes - 4
H*nmm*		Hex data in fixed-length field of $*nmm* bytes
HEXS	•	Hex dump to next skip or sizeof end
HEXD	•	All data to end of resource or unknown keyed item

Structure Changes for Existing Resources

+BYT	•	Insert a byte when opening
+WRD	•	Insert a word when opening
+LNG	•	Insert a long when opening
+*nmm*	•	Insert $*nmm* bytes when opening
+PST	•	Insert an empty Pascal string when opening
+EST	•	Insert an empty even-padded Pascal string
+CST	•	Insert an empty C string when opening
-BYT	•	Delete a byte when closing
-WRD	•	Delete a word when closing
-LNG	•	Delete a long when closing
-*nmm*	•	Delete $*nmm* bytes when closing
-PST	•	Delete a Pascal string when closing
-EST	•	Delete an even-padded Pascal string when closing
-CST	•	Delete a C string when closing.

Filler (uneditable)

FBYT		Filler byte
FWRD		Filler word
FLNG		Filler long
F*nmm*	•	$*nmm* bytes of filler

EVEN AND ODD BYTE BOUNDARIES

As far as the Data Editor is concerned, the bytes in a field can begin at any byte offset in the resource, regardless of whether that offset is even or odd. Most compilers and hardware, however, require a variety of objects (e.g. long words, floats, structures, etc.) to begin on even or sometimes longword memory addresses. In general, it is a good idea to attempt to make all fields start at an even offset into the resource data to avoid problems.

When your application reads a resource into memory, the Mac's Resource Manager guarantees that the 0'th byte of the resource falls on an even (in fact, a multiple of 16 bytes) address in main memory. Thus it is sufficient to look only at the even or odd parity of any field's offset from the beginning of its resource data to know whether the start of that field's data will fall on an even or odd address.

Note: You can see the current field offsets when you edit your custom resource. They are made visible when you choose the **Show Field Offsets** command from the Data Editor's **Field** menu. They are normally shown in hexadecimal, but the **Decimal Offsets** command converts them to decimal.

EXPLANATIONS OF FIELD TYPES

The Data Editor supports the following field types, listed alphabetically by type. These types are all found in the popup menu of the Template Editor's field editing window.

Note: Types marked here and in the popup menu with a • are not supported by Apple's old ResEdit program.

+BYT • Insert a Byte when Opening
+WRD • Insert a Word when Opening
+LNG • Insert a Long when Opening

Each of these fields inserts one (+BYT), two (+WRD), or four (+LNG) zero-valued bytes at the position in the resource the field represents in the template. The newly inserted data is marked to show the change.

+PST • Insert a Pascal String
+CST • Insert a C String

Each of these fields inserts an empty Pascal (+PST) or C (+CST) string at the position in the resource that the field represents in the template. In both cases, this is a single zero-valued byte. The newly inserted empty string is marked to show the change.

+EST • Insert an Even-Padded Pascal String

This inserts an empty Pascal (or C) string at the position in the resource that the field represents in the template. The data inserted is a single zero-valued byte followed by either zero or one pad bytes to ensure that the following field begins on an even byte offset. The newly inserted empty string is marked to show the change.

+*nmm* • Insert Fixed-Length Block of Bytes

This inserts $*nmm* zero-valued bytes at the position in the resource that the field represents in the template. The length of the block is $*nmm* bytes, where the first hex digit, *n*, is in the range '0' to '9', and the following two hex digits, *mm*, are each in the range '0' to 'F' (upper case only). The field type for the minimum insertion block size is +000, which inserts nothing; the maximum is +9FF, or 2559 (decimal) bytes.

-BYT • Delete a Byte when Closing
-WRD • Delete a Word when Closing
-LNG • Delete a Long when Closing

Each of these fields deletes one (-BYT), two (-WRD), or four (-LNG) bytes at the position in the resource the field represents in the template. The data to be deleted is parsed when you open the resource with Data Editor, and is marked to show the deletion that will occur when you save the resource.

-PST • Delete a Pascal String

The -PST field marks for deletion a Pascal string at the position in the resource that the field represents in the template. The length of the data deleted is 1 + *n* bytes, where *n* is the unsigned value, ranging from 0 to 255, of the byte at the current parsing offset.

-CST • Delete a C String

The -CST field marks for deletion a C string at the position in the resource that the field represents in the template. The length of the data deleted is 1 + *n* bytes, where *n* is the number of non-null bytes extending from the current byte parsing offset to the first null byte.

-EST • Delete an Even-Padded Pascal String

The -EST field marks for deletion a padded Pascal string at the position in the resource that the field represents in the template. The length of the data deleted is 1 + *n* + *pad* bytes, where *n* is the unsigned value, ranging from 0 to 255, of the byte at the current parsing offset, and *pad* is either 0 or 1 byte, depending on whether the following field starts at an even offset or not.

-*nmm* • Delete a Fixed-Length Block of Bytes

The -*nmm* field marks for deletion $*nmm* bytes at the position in the resource that the field represents in the template. The length of the block is taken to $*nmm* bytes, where the first hex digit, *n*, is in the range '0' to '9', and the following two hex digits, *mm*, are each in the range '0' to 'F' (upper case only). The field type for the minimum deletion block size is +000, which deletes nothing; the maximum is +9FF, or 2559 (decimal) bytes. The bytes are deleted when you save the resource.

AWRD - Align Next Field on Word Boundary
ALNG - Align Next Field on Long Boundary
AL08 • Align next field on 8-byte Boundary
AL16 • Align next field on 16-byte Boundary

Each of these fields appends 0 or more null pad bytes after the previous field, in order to guarantee that the next field begins at an offset into the resource that is divisible by 2 (AWRD), 4 (ALNG), 8 (AL08), or 16 (AL16). The number of pad bytes added is whatever it takes to guarantee that the following field always starts with the proper alignment. You will usually want to use one of these fields after a variable-length field. These fields are not editable.

BBIT - Byte Bit
BB*nn* • Byte Bit Field

The BBIT and BB*nn* fields occur in groups specifying individual bits or groups of bits within a single byte of data (8 bits). The BB*nn* field declares a bit field *nn* bits wide (01 <= *nn* <= 08) anywhere within the byte, as long as the field does not cross the byte's boundary. As BBIT and BB*nn* fields are encountered in order, they specify bits beginning with the most significant high-order bit of the byte. When the Data Editor displays each bit or bit field, the field is labeled with the bit number or numbers (inclusive) that define the bit field. The value of a single bit is shown as either "On" if the bit is set, or "Off" if the bit is clear. The value of a bit field is numeric. You can change the value of single bits using the Data Editor's **Toggle Value** menu command.

BCNT • Byte Count of List Items

The BCNT field is 1 byte long and contains an unsigned integer, *n*, in the range from 0 to 255. This number represents how many repeated items will be found in the first counted list that occurs anywhere after the BCNT field at the same item nesting level. Counted lists begin with an LSTC field. As you add or delete items from the list, the BCNT field is automatically updated for you; it is otherwise uneditable.

The label for this field should be the plural name of the items to be counted, such as "Networks" or "Options". For best results, do not use labels that start out "Number of" or "# of" (such as "Number of networks" or "# of Options") since the Data Editor copies the value of the label as a header string for each item in the displayed list.

BFLG • Byte Flag

The BFLG field displays the value of the low-order bit 0 of a byte. The other bits are inaccessible and set to 0. This field is analogous to a C language `char` variable that you are using as a boolean flag. You can quickly change the flag's value using the Data Editor's **Toggle Value** menu command.

BHEX • Byte-Length Hex Data

The BHEX field is a variable length field containing a single byte followed by a block of data of unknown format. The first byte of the field contains the length, *n*, of the block, and the following *n* bytes are the pure hex data, where *n* is in the range 0 to 255. The initial byte's value does not include its own length. This field can start and end at any byte offset, and is always 1+*n* bytes in length. It is usually a good idea to follow BHEX fields with an AWRD or ALNG alignment field. The initial length byte is not displayed but is computed automatically.

BOOL - Boolean

The BOOL field is 2 bytes long and is used to encode a Boolean value, which is displayed as "TRUE" if either byte is non-zero, and "FALSE" if both bytes have 0 in them. You can change the value of a BOOL field using the Data Editor's **Toggle Value** menu command, as well as by opening its value dialog. If the value is set to TRUE, both bytes are written out with the default value $0100. This field can begin at any byte offset in the resource; however, even offsets are strongly recommended.

Note: If your development language requires Boolean fields to be in a different format from the above, you can change the value to any bit pattern you want by editing Resorcerer's 'BOOL' 128 resource, which contains the bit pattern the editor uses to represent TRUE when writing BOOL fields back out.

BSHX • Byte-Skip Hex Data

The BSHX field is a variable length field containing a single byte followed by a block of *n* bytes of data of unknown format. The first byte of the field contains the value, *1+n*, which is the size of the entire field, including the initial byte. *n* is in the range 1 to 255.

This field can start and end at any byte offset, and is always 1+*n* bytes in length. It is usually a good idea to follow BSHX fields with an AWRD or ALNG alignment field. The initial length byte is not displayed but is computed automatically.

BSIZ • Byte Sizeof

The BSIZ field is 1 byte long, and contains an unsigned decimal value between 0 and 255 that represents the number of bytes in the following data, up to a matching SKPE field. Typically, intervening fields contain complex structures of variable size or unknown format. The byte count stored in the BSIZ field does not include its own size.

BSIZ...SKPE pairs may not be nested except as part of the standard nesting of repeated list or keyed items. BSIZ fields are not editable, but their values are automatically computed for you. Their values are only shown when you have the **Show Offsets** option set in the Data Editor.

We discourage the use of BSIZ fields in favor of BSKP (or better yet, LSKP) fields, which are used more often in many Apple resources.

BSKP • Byte Skip Offset

The BSKP field is 1 byte long, and contains an unsigned decimal value between 1 and 255 that represents the number of bytes, including the BSKP's byte, to skip over to get to a matching SKPE field. Typically, intervening fields contain complex structures of variable size; skip offsets allow for fast scanning algorithms when your resource data has complex variable sized items or fields in it that you need to skip over quickly.

BSKP...SKPE pairs may not be nested except as part of the standard nesting of repeated list or keyed items. BSKP fields are not editable, but are automatically set for you. Their values are only shown when you have the **Show Offsets** option set in the Data Editor.

BSTR • Byte Length Text String

The BSTR field is a variable-length field for displaying a string of text. This field is the same as the PSTR field, but is included here for completeness. The first byte of the field contains the length, *n*, of the string, and the following *n* bytes are the characters in the string, where *n* is in the range 0 to 255. If there are an even number of characters in the string, the total field length will be odd (if the

BSTR field begins on an even byte boundary, and has an odd number of bytes in it, the field after it will begin on an odd boundary). This field can start and end at any byte offset, and is always at least 1 byte in length.

The Data Editor displays non-empty strings using double quotes.

CASE • Symbolic and/or Default Value for Previous Data Field

The CASE field is a special field type that lets you define symbolic values for a previous non-CASE field in the 'TMPL' resource. Any number of CASE fields can appear together in sequence, with all of them refering to the previous non-CASE field in the TMPL resource. When the Template Editor displays CASE fields, they are indented so that you can easily pick out the previous field to which they refer.

The label string for a CASE field defines the symbolic value using the simple syntax

<*name*> = <*valuestring*>

where <*name*> is any non-empty sequence of ASCII characters up to but not including the '=' character, followed by <*valuestring*> which consists of everything past the first '=' to the end of the label string.

When the Data Editor displays or opens a field that has symbolic cases associated with it, all symbolic value names are collected and used to create a popup menu from which you can choose various values to be installed in the field. This menu is available at all times in the Data Editor's field list as a small popup box to the left of the field label; it is also installed in the field's editing dialog when you open the field.

Sorcery: When a field has exactly one CASE in its list of symbols, and its value is the same as that case, the Data Editor suppresses the popup menu in the field list, since there is nothing useful to do with it. To change the value of the field to something other than a defined CASE, you will have to open it for editing.

Whenever the Data Editor creates new resource data, either as the resource is created or when you create a new item in a list or new keyed item, the field must be set to some default value. Normally this is 0 or other value signifying empty or missing. However, if the data field has at least one CASE field, the value of the first CASE is taken as the default. This lets you avoid situations where the Editor's default value may be illegal (consider, for example, a

numeric scaling factor field that should never be 0). Thus, your most common symbolic value for a field should be its first CASE.

Resources which begin with a version field are prime candidates for having a single CASE declaring the value of the latest version.

CHAR - Single Character

The CHAR field is 1 byte long and encodes a single ASCII character that the Data Editor displays surrounded by single quotes. If the character value is less than $20, or equal to $7F, the Editor will display its value in hex. You can enter any character's value in its value-editing dialog as either a single character, or as a 3-character string starting with a '$' and followed by exactly two hex digits.

CHAR fields in new resource data are initially set to single blanks.

CODE - Code Dump

The CODE field represents 0 or more bytes of trailing data in the resource that should be disassembled as 68040 machine instructions. The Data Editor displays only the first disassembled instruction, if any, until you open the field for editing.

COLR • Color RGB Triplet

The COLR field is 6 bytes long and encodes a standard QuickDraw RGBColor structure (as documented in Inside Macintosh, volume 5, chapter 4) consisting of three unsigned 2-byte words, one for each primary component. The Data Editor displays the value of the COLR field as

(r,g,b) = (0,65535,23434) (or whatever the numbers are)

so that there is no ambiguity as to which component is which. It also displays a small swatch of the color next to the numeric components. Each component number must be in the range 0 to 65535. This field can start at any byte boundary in the resource; however, even boundaries are strongly recommended.

When you open the value of a COLR field, you can set the color component values directly, or click on the **Set** button to use the Standard Color Picker.

CSTR - C String

The CSTR field is a variable length field containing a C string, an

arbitrarily long sequence of non-null ASCII characters followed by a null (0 byte). This field can start and end at any byte offset, and is always at least 1 byte in length. If there are an even number of characters in the string, the total length of the field will be odd, and vice-versa.

C*nmm* - Fixed Length C String

The C*nmm* field is a fixed-length block of bytes in which to place a C string, beginning at byte 0 of the block. The length of the block is taken to be $*nmm* bytes, where the first hex digit, *n*, is in the range '0' to '9', and the following 2 hex digits, *mm*, are each in the range '0' to 'F' (upper case only). This field can begin at any byte offset. The minimum block size for a C string is C001, since at least one null byte is required for the empty C string. The maximum block size encodable using this scheme is C9FF, or 2559 (decimal) bytes, including the trailing null byte. Bytes in the field beyond the logical end of the C string are set to 0.

DATE • System Date and Time

The DATE field is 4 bytes long and holds a system date/time number, which is a count of the number of seconds since Jan. 1, 1904. The Data Editor displays values of the field as a readable string indicating the date and time the value represents. This field can start at any byte offset in the resource; however, even offsets are strongly recommended.

When the Data Editor creates a DATE field in a new resource, it fills the field in with the value of the current date and time.

DBYT - Decimal Byte

The DBYT field is 1 byte long and encodes a signed decimal number in the range -128 to 127. This field's value-editing dialog will also accept hex input preceded by a '$'.

DLNG - Decimal Long

The DLNG field is 4 bytes long and encodes a signed decimal number in the range -2,147,483,648 to 2,147,483,647. The field can begin at any byte offset in the resource; however, even offsets are strongly recommended. This field's value-editing dialog will also accept hex input preceded by a '$'.

DOUB • Double Precision 64-bit Floating Point Number

The DOUB field is 8 bytes long and encodes a double precision floating point number. In THINK C, such a number is declared as a `short double`.

You can append a formatting command to the end of this field's label string. The syntax of the command is similar to a standard C language `printf` floating point escape sequence: *%n*. *n* is the number of significant digits the Data Editor will use to display the floating point value; if the field label string does not exist, or is illegal, the Editor uses a default value.

DVDR - Divider Comment Line and/or Section Label

The DVDR field does not represent data. It is simply a way of visually separating and marking sections of related data within a resource or item. The field label is drawn on the right side of the Data Editor's list to better make the label stand out. DVDR fields are helpful when editing complex resources, such as MacApp 'view's.

DWRD - Decimal Word

The DWRD field is 2 bytes long and encodes a signed decimal number in the range -32768 to 32767. The field can begin at any byte offset in the resource; however, even offsets are strongly recommended. This field's value-editing dialog will also accept hex input preceded by a '$'.

ECST - Even C String

The ECST field is a variable length field containing a C string, an arbitrarily long sequence of non-null ASCII characters followed by a null (0 byte). This field can start and end at any byte offset, and is always at least 2 bytes in length. If the length of the string is found to be even, an extra pad (zero) byte is appended after the terminating null byte to ensure that the entire field uses up an even number of bytes (if an ECST field begins at an odd byte offset, the following field will also).

ESTR - Even Pascal String

The ESTR field is a variable length field containing a Pascal string. The first byte of the field contains the length, *n*, of the string, and

the following *n* bytes are the characters in the string, where *n* is in the range 0 to 255. If *n* is even (thereby making the total field length odd), an extra zero pad byte is appended to ensure that the total field length is even (if an ESTR field begins at an odd byte offset, the following field will also). This field can start and end at any byte offset, and is always at least 2 bytes in length.

The ESTR field is equivalent to a PSTR field immediately followed by an an AWRD field.

EXTN • Extended Precision 80-bit Floating Point Number

The EXTN field is 10 bytes long and encodes a SANE extended precision 80-bit floating point number. In THINK C, such a number is declared as either `extended` or `double`, depending on the compiler settings. This field is the same as the XT80 field.

You can append a formatting command to the end of this field's label string. The syntax of the command is similar to a standard C language `printf` floating point escape sequence: *%n*. *n* is the number of significant digits the Data Editor will use to display the floating point value; if the field label string does not exist, or is illegal, the Editor uses a default value.

FBYT - Filler Byte

The FBYT field is 1 byte long and is used to indicate an unused or reserved byte in the resource data. Its value is displayed in the range -128 to 127, however FBYT fields are not editable.

FCNT • Fixed Counted List

The FCNT field does not declare or parse any actual data. It is analogous to the other item count fields, such as OCNT, BCNT, LCNT etc. It must be followed somewhere at the same indentation level by the LSTC item record start field. The first number in the field label string, either decimal or hex, is used as the list count to parse that many items in the data. The count can be any non-negative 32-bit number. Since the count is embedded in the template, not the data, the list size is fixed.

During editing, the Data Editor does not enforce the array length, so that you can cut and paste items to rearrange them. The FCNT field is automatically updated the same as the other field types if you do change the array length, but its displayed value is not added to the

resource data when you close the resource. When you attempt to build the resource data during a save or duplication, the Editor warns you about any discrepancy between the current number of repeated items and the declared number. If you continue to build the resource data with a different number of items in the list, the data will no longer be described by your current template until you change the item count in the FCNT field's label to match.

The label for this field should be the plural name of the items to be counted, such as "Networks" or "Options". For best results, do not use labels that start out "Number of" or "# of" (such as "Number of networks" or "# of Options") since the Data Editor copies the value of the label as a header string for each item in the displayed list.

FIXD • Fixed Point Number

The FIXD field is 4 bytes long and encodes a standard `Fixed` number in the range [-32768.0, 32768.0). The high-order word contains the integer part; the low-order word contains the fractional part. This field can start at any byte offset in the resource; however, even offsets are strongly recommended.

FLNG - Filler Long

The FLNG field is 4 bytes long and is used to indicate an unused or reserved long in the resource data. Its value is displayed in the range -2,147,483,648 to 2,147,483,647. FLNG fields are not editable. The field can begin at any byte offset in the resource; however, even offsets are strongly recommended.

FLTR • This is a Filtered Template

The FLTR field declares no data, but should appear at the start of any filtered template. Its presence tells the Data Editor that the resource data to be edited should be filtered on input and output to convert the data into and out of the form that the filtered template fields in the rest of the template describe. The field label serves to remind you that what you are editing does not have the same structure as what you might otherwise expect. Filters are explained more fully in the "Designing Filters" section later in this chapter.

FRAC • Fract Number

The FRAC field is 4 bytes long and encodes a Macintosh `Fract` number in the range [-2.0, 2.0). The high-order 2 bits of the field are

the signed integer part; the low-order 30 bits are the high-precision fractional part. This field can start at any byte offset in the resource; however, even offsets are strongly recommended.

FWID • Font Width Fixed Point Number

The FWID field is 2 bytes long and encodes a short fixed point number in the range [-8.0, 8.0). The upper 4 bits of the field encode the signed integer portion of the number; the lower 12 bits the fractional portion. This field can start at any byte offset in the resource; however, even offsets are strongly recommended.

FWRD - Filler Word

The FWRD field is 2 bytes long and is used to indicate an unused or reserved word in the resource data. Its value is displayed in the range -32768 to 32767. FWRD fields are not editable. The field can begin at any byte offset in the resource; however, even offsets are strongly recommended.

F*nmm* - Filler Block

The F*nmm* field is a fixed-length block of uneditable filler bytes. The length of the block is taken to be $*nmm* bytes, where the first hex digit, *n*, is in the range '0' to '9', and the following 2 hex digits, *mm*, are each in the range '0' to 'F' (upper case only). This field can begin at any byte offset. The minimum block size is the empty block, F000. The maximum block size encodable using this scheme is F9FF, or 2559 (decimal) bytes.

HBYT - Hex Byte

The HBYT field is 1 byte long and encodes an unsigned hexadecimal number in the range $00 - $FF. When editing a value of this type, an initial '$' is optional and the hex digits can be in either upper or lower case.

HEXD • Hex Dump of Unknown Data

The HEXD field represents 0 or more bytes of trailing data in the resource that for whatever reasons cannot be encoded using the other field types (typically, the resource is allowed to have optional data appended to it after the last required field).

The HEXD field is also used as the sole item in the last keyed item of

a keyed item list that is surrounded by a skip offset pair. For more on this, see the explanations for keyed items (KEYB, KEYE, etc.).

HEXS • Sized Hex Dump

The HEXS field consists of 0 or more bytes of arbitrary data to be displayed as hex. The HEXS field must explicitly or implicitly be followed by a SKPE field, so that the parser can know when to stop parsing data and start parsing subsequent fields. The amount of data parsed is equal to the value of the previous matching skip offset (BSKP, WSKP, or LSKP) or sizeof (BSIZ, WSIZ, or LSIZ) in the resource, minus the lengths of any intermediate fields.

The HEXS field lets you do a local dump of a variable length of data up to a given length recorded earlier in the resource. For example, if in the sequence "WSIZ, REAL, REAL, REAL, HEXS, SKPE" the value of the WSIZ field were 32, then the HEXS field would be 32 - 3*4 bytes of text, since REAL fields are 4 bytes long.

HLNG - Hex Long

The HLNG field is 4 bytes long and encodes an unsigned hexadecimal number in the range $00000000 - $FFFFFFFF. When editing a value of this type, an initial '$' is optional and the hex digits can be in either upper or lower case. The field can begin at any byte offset in the resource; however, even offsets are strongly recommended.

HWRD - Hex Word

The HWRD field is 2 bytes long and encodes an unsigned hexadecimal number in the range $0000 - $FFFF. When editing a value of this type, an initial '$' is optional and the hex digits can be in either upper or lower case. The field can begin at any byte offset in the resource; however, even offsets are strongly recommended.

H*nmm* - Fixed Length Hex Data

The H*nmm* field is a fixed length block of bytes in which to place pure data, beginning at byte 0 of the block. The data is editable as hex only. The length of the block is taken to be $*nmm* bytes, where the first hex digit, *n*, is in the range '0' to '9', and the following 2 hex digits, *mm*, are each in the range '0' to 'F' (upper case only). This field can begin at any byte offset. The minimum block size is H000, that is, the empty block, which allocates 0 bytes of storage. The

maximum block size encodable using this scheme is H9FF, or 2559 (decimal) bytes. Bytes in the field beyond the logical end of the hex data you enter into it are set to 0.

KBYT • Byte Key

The KBYT field contains a 1-byte signed decimal number in the range -128 to 127. The field declares the beginning of a keyed item list in the template, as discussed more fully in the following section on KEYB and KEYE fields.

Newly created key fields take on the value of their first CASE.

KCHR • Character Key

The KCHR field contains a 1-byte character, which, like the CHAR field, can be specified by either a single printable character, or by a hex pair preceded by a '$' sign. The field can begin at any offset in the resource data. The field declares the beginning of a keyed item list in the template, as discussed more fully in the following section on KEYB and KEYE fields.

Newly created key fields take on the value of their first CASE.

KEYB • Begin Keyed Item
KEYE • End of Keyed Item

Key (KBYT, KWRD, KLNG, KCHR, and KTYP) fields each declare the start of a *keyed item list* in the TMPL. When scanning your resource data, the value of any key field determines which of several formats a section of subsequent data in the resource is about to be found in. Each of these possible formats is kept as an item in the key's item list. The Data Editor uses the key field value to determine which keyed item format to use to parse and display the data. If you are used to creating Rez template source code, you will recognize this as the equivalent of a Rez `switch` statement, where each keyed item represents the format of a particular `case`.

A keyed item list consists of the key field followed by 1 or more keyed items, where each keyed item begins with the KEYB field and ends with a matching KEYE field. The fields declared between the KEYB and KEYE fields can be any sequence of 0 or more template fields that describes the structure of the keyed item (unlike items in repeated lists, keyed items can be empty). The data fields comprising the item can include nested key fields with their own

item lists, or nested lists of repeated items, etc., up to some maximum nesting level. There can be no fields declared between the KEYE field of an item and the KEYB field of the next item in the list; any non-KEYB field following a KEYE field signals the end of the keyed item list.

In addition to the list of keyed items, a key field must have one or more symbolic CASE fields, one for each key value to be associated with a particular KEYB field in the keyed item list. If there are N CASEs for the key, then there must be N keyed items (that is, KEYB...KEYE pairs) following the key's last CASE field (for a clearer idea of how this works, see the Examples section later in this chapter).

The linkage between cases and keyed items is done by indexing into the list of alternate formats. That is, the *i*'th CASE is associated with the *i*'th KEYB in the list. For future compatibilty, however, the label of each KEYB field should be an exact copy of the <*valuestring*> of its corresponding CASE. You will want to do this anyway to help identify each keyed item.

When new resource data is created, the first CASE in a key's list is used as the value assigned to the key field, and the keyed item associated with the first CASE is used as the new keyed item.

For a standard keyed item list, it is illegal for the value of the key (KBYT, KWRD, KLNG, KCHR, KTYP, ...) field as found in the resource data to be anything other than one of the values represented in the key's list of CASEs. The Data Editor won't let you change the field value to anything other than one of the values in its list of CASEs; however, if some other value is found during the opening of the resource, the Editor will complain and be unable to continue, since it cannot identify the correct format that the data following the unknown key is about to be found in.

However, if you design your resource so that keyed item lists are surrounded by a skip offset pair of fields (e.g. SKIP...SKPE), then the Editor can find the end of the keyed item even though the alternate format and its unknown key are not represented in the keyed item list. To do this, there must be one extra keyed item (that is, a KEYB...KEYE pair) appended to the N keyed items that correspond to the N CASEs of the key. This final extra keyed item should have between the KEYB and KEYE fields exactly one HEXD field, which represents the unknown or default format.

KHBT • Hex Byte Key

The KHBT field contains a 1-byte unsigned hex number in the range $00 to $FF. The field declares the beginning of a keyed item list in the template, as discussed in the section on KEYB and KEYE fields.

Newly created key fields take on the value of their first CASE.

KHLG • Hex Long Key

The KHLG field contains a 4-byte unsigned hex number in the range $00000000 to $FFFFFFFF. The field can begin at any byte offset in the resource; however, even offsets are strongly recommended. The field declares the beginning of a keyed item list in the template, as discussed in the section on KEYB and KEYE fields.

Newly created key fields take on the value of their first CASE.

KHWD • Hex Word Key

The KHWD field contains a 2-byte unsigned hex number in the range $0000 to $FFFF. The field can begin at any byte offset in the resource; however, even offsets are strongly recommended. The field declares the beginning of a keyed item list in the template, as discussed in the section on KEYB and KEYE fields.

Newly created key fields take on the value of their first CASE.

KLNG • Long Key

The KLNG field contains a 4-byte signed decimal number in the range -2,147,483,648 to 2,147,483,647. The field can begin at any byte offset in the resource; however, even offsets are strongly recommended. The field declares the beginning of a keyed item list in the template, as discussed more fully in the previous section on KEYB and KEYE fields.

Newly created key fields take on the value of their first CASE.

KTYP • Type Key

The KTYP field contains a 4-byte character type name, which, like the TNAM field, can be specified by either the four characters directly, or by four pairs of hex digits preceded by a '$' sign. The

field can begin at any offset in the resource data; however, even offsets are strongly recommended. The field declares the beginning of a keyed item list in the template, as discussed more fully in the previous section on KEYB and KEYE fields.

Newly created key fields take on the value of their first CASE.

KUBT • Unsigned Decimal Byte Key

The KUBT field contains a 1-byte unsigned decimal number in the range 0 to 255. The field declares the beginning of a keyed item list, as discussed in the section on KEYB and KEYE fields.

Newly created key fields take on the value of their first CASE.

KULG • Unsigned Decimal Long Key

The KULG field contains a 4-byte unsigned decimal number in the range 0 to 4,294,967,295. The field can begin at any byte offset in the resource; however, even offsets are strongly recommended. The field declares the beginning of a keyed item list in the template, as discussed in the section on KEYB and KEYE fields.

Newly created key fields take on the value of their first CASE.

KUWD • Unsigned Decimal Word Key

The KUWD field contains a 2-byte unsigned decimal number in the range 0 to 65,535. The field can begin at any byte offset in the resource; however, even offsets are strongly recommended. The field declares the beginning of a keyed item list in the template, as discussed in the section on KEYB and KEYE fields.

Newly created key fields take on the value of their first CASE.

KWRD • Word Key

The KWRD field contains a 2-byte signed decimal number in the range -32768 to 32767. The field can begin at any byte offset in the resource; however, even offsets are strongly recommended. The field declares the beginning of a keyed item list in the template, as discussed more fully in the previous section on KEYB and KEYE fields.

Newly created key fields take on the value of their first CASE.

LBIT • Long Bit
LBnn • Long Bit Field

The LBIT and LB*nn* fields occur in groups specifying individual bits or groups of bits within a single long word of data (32 bits). The LB*nn* field declares a bit field *nn* bits wide (01 <= *nn* <= 32) anywhere within the long word, with the proviso that the field does not cross the long word's boundary. As LBIT and LB*nn* fields are encountered in order, they specify bits beginning with the most significant bit of the word. When the Data Editor displays each bit or bit field, the field is labeled with the bit number or numbers (inclusive) that define the bit field. The value of a single bit is shown as either "On" if the bit is set, or "Off" if the bit is clear. The value of a bit field is numeric. You can change the value of single bits using the Data Editor's **Toggle Value** menu command. The group of 32 bits can begin at any byte offset in the resource; however, even offsets are strongly recommended.

LCNT • Long Count of List Items

The LCNT field is 4 bytes long and contains an unsigned integer, *n*, in the range from 0 to 4,294,967,295. This number represents how many items are repeated in the first counted list beginning with a LSTC field that occurs anywhere after the LCNT field at the same item nesting level. As you add or delete items from the list, this field is automatically updated for you; it is otherwise uneditable.

The label for this field should be the plural name of the items to be counted, such as "Networks" or "Options". For best results, do not use labels that start out "Number of" or "# of" (such as "Number of networks" or "# of Options") since the Data Editor copies the value of the label as a header string for each item in the displayed list.

LFLG • Long Flag

The LFLG field displays the value of the low-order bit 0 of a long. The other 31 bits are inaccessible and set to 0. You can change the flag value using the Data Editor's **Toggle Value** menu command.

LHEX • Long Count of Pure Hex Data

The LHEX field is a variable length field containing a 4-byte long followed by a block of data of unknown format. The first 4 bytes of the field contain the length, *n*, of the block, and the following *n* bytes

are the pure hex data, where *n* is in the range 0 to 4,294,967,295. This field can start and end at any byte offset, and is always 4+*n* bytes in length. It is usually a good idea to follow LHEX fields with an AWRD or ALNG alignment field. The initial length field is not displayed, but is automatically computed for you.

LSHX • Long Skip Hex Data

The LSHX field is a variable length field containing a 4-byte long followed by a block of hex data of unknown format. The first 4 bytes of the field contain the skip offset, *n+4*, of the block, and the following *n* bytes are the pure hex data, where *n* is in the range 0 to 4,294,967,291. That is, the field value includes its own size. This field can start and end at any byte offset, and is always 4+*n* bytes in length. It is usually a good idea to follow LHEX fields with an AWRD or ALNG field. The initial skip offset is not displayed, but is computed automatically for you.

LSIZ • Long Sizeof

The LSIZ field is 4 bytes long, and contains an unsigned decimal value between 0 and 4,294,967,295 that represents the number of bytes in the following data, up to a matching SKPE field. Typically, intervening fields contain complex structures of variable size or unknown format. The byte count stored in the LSIZ field does not include its own size.

LSIZ...SKPE pairs may not be nested except as part of the standard nesting of repeated list or keyed items. LSIZ fields are not editable (their values are automatically computed for you), and their values are only shown when you have the **Show Offsets** option set in the Data Editor.

We discourage the use of LSIZ fields in favor of LSKP fields, which are more elegant and used more often in Apple resources.

LSKP • Long Skip Offset

The LSKP field is 4 bytes long and contains an unsigned decimal long value between 4 and 4,294,967,295, which represents the number of bytes, including the 4 bytes of the LSKP field itself, to skip to get to a matching SKPE field. Typically, intervening fields contain complex structures of variable size; skip offsets allow for fast scanning algorithms when your resource data has complex variable sized items or fields in it that you need to skip over quickly.

Although this field can occur at any byte offset in the resource, even offsets are strongly recommended.

LSKP...SKPE pairs may not be nested except as part of the standard nesting of repeated list or keyed items. LSKP fields are not editable, but are automatically computed for you. Their values are only shown when you have the Data Editor's **Show Offsets** option set.

LSTB - Begin a Non-Counted List Item

The LSTB field indicates the start of a series of 'TMPL' fields that together describe the structure of a repeatable list item in the resource. The length of the list is not explicitly saved in the resource data and must be determined by whatever means at runtime, such as comparing the beginning of the list with the size of the resource. The LSTB field itself does not represent any data in the resource. A matching LSTE field must occur later on in the template to indicate the end of the item description. The label string is ignored.

The only way to tell when the end of a non-counted list occurs is by knowing beforehand what the size of the resource is. For this reason, you cannot specify data fields after the end of a non-counted list in your TMPL resource. Also, non-counted lists cannot themselves be nested as an item in some other list.

LSTC - Start of Counted List Item

The LSTC field indicates the start of a set of fields that together describe the structure of a repeatable list item. It must occur in your template field list at some point after a OCNT, BCNT, LCNT, FCNT, or ZCNT field occuring at the same item nesting level. The LSTC field does not represent any data in the resource. A matching LSTE field must occur later on in the TMPL to indicate the end of the item description. List items can themselves contain lists or keyed items, nested to some maximum depth. The label string is ignored.

LSTE - End of List Item

The LSTE field signifies the end of a series of TMPL fields that describe the structure of a repeatable list item. It must match some previous LSTC, LSTB, or LSTZ field. There must be one LSTE field for every nested list item description in the TMPL. The field itself does not represent any data in the resource, unless it matches a LSTZ field, in which case it causes a null byte to be added after the last item in the list. The label string is ignored.

LSTR - Long String

The LSTR field is a variable length field containing a 4-byte word followed by a string of ASCII characters. The first 4 bytes of the field contain the length, *n*, of the string, and the following *n* bytes are the characters in the string, where *n* is in the range 0 to 4,294,967,295. This field can start and end at any byte offset, and is always 4+*n* bytes in length. Note that if there are an odd number of characters in the string, the total length will be odd (if the LSTR field begins on an even byte boundary, and has an odd number of bytes in it, the field after it will begin at an odd offset). It is usually a good idea to follow LSTR fields with an AWRD or ALNG field.

The Data Editor can only edit up to 32K bytes per field.

LSTS - Begin a Sized List Item

The LSTS field indicates the start of a series of 'TMPL' fields that together describe the structure of a repeatable list item. The LSTS field itself does not represent any data in the resource. The length of the sized list is determined by an enclosing sizeof or skip offset (BSKP, WSKP, LSKP...SKPE) to indicate where in the resource the first field after the last item in the list starts. This will be at the following SKPE field at the same nesting level as the list. Before the SKPE field a matching LSTE field must occur to indicate the end of the item description. The label string is ignored.

LSTZ - Begin a List Item, with Final Item Followed by a 0 Byte

The LSTZ field indicates the start of a series of TMPL fields that together describe the structure of a repeatable list item in the resource. The length of the list is not explicitly saved in the resource data; a byte with a 0 in it is appended after the last item in the list. The LSTZ field itself does not represent any data in the resource. A matching LSTE field must occur later on in the TMPL to indicate the end of the item description. List items can themselves contain lists, nested to some maximum depth. The label string is ignored.

This field type, originally designed to parse 'MENU' resources, is included for compatibility with ResEdit; however, its use is strongly discouraged. If the first field of the repeatable item (that is, the template field following this LSTZ field) has a 0 in it, your resource parsing code may get into trouble.

LZCT • Long Zero-based Count of List Items

The LCNT field is 4 bytes long and contains an signed integer, *n*, in the range from -1 to 2,147,483,647. This number represents 1 less than the number of repeated items in the first counted list beginning with a LSTC field that occurs anywhere after the LZCT field at the same item nesting level. As you add or delete items from the list, this field is automatically updated for you; it is otherwise uneditable. This field type is included for completeness, but we discourage its use. Use the one-based counts (BCNT, OCNT, or LCNT) instead.

The label for this field should be the plural name of the items to be counted, such as “Networks” or “Options”. For best results, do not use labels that start out "Number of” or “# of” (such as “Number of networks” or “# of Options”) since the Data Editor copies the value of the label as a header string for each item in the displayed list.

MDAT • Modification Date and Time

The MDAT field is 4 bytes long and, like the DATE field, holds a system date/time number, which is a count of the number of seconds since Jan. 1, 1904. The Data Editor displays values of the field as a readable string indicating the date and time the value represents. This field can start at any byte offset in the resource; however, even offsets are strongly recommended.

When the Data Editor saves an MDAT field in a changed or new resource, it sets the field to the value of the current date and time.

OCNT • One-based Word Count

The OCNT field is 2 bytes long and contains an unsigned integer, *n*, in the range from 0 to 65535. This number represents how many items are repeated in the first counted list beginning with a LSTC field that occurs anywhere after the OCNT field at the same item nesting level. As you add or delete items from the list, this field is automatically updated for you; it is otherwise uneditable.

The label for this field should be the plural name of the items to be counted, such as “Networks” or “Menus”. For best results, do not use labels that start out "Number of” or “# of” (such as “Number of networks” or “# of Menus”) since the Data Editor copies the value of this label as a header string for each item in the displayed list.

OCST - Odd C String

The OCST field is a variable-length field containing a C string, an arbitrarily long sequence of non-null ASCII characters followed by a null (0 byte). This field can start and end at any byte offset, and is always at least 1 byte in length. If the length of the string is found to be odd, an extra pad (zero) byte is appended after the terminating null byte to ensure that the entire field uses up an odd number of bytes (if an OCST field begins at an even byte offset, the following field will start at an odd offset, and vice-versa).

This field type is included for compatibility with ResEdit; we strongly discourage using the OCST field.

OSTR - Odd Pascal String

The OSTR field is a variable-length field containing a Pascal string. The first byte of the field contains the length, *n*, of the string, and the following *n* bytes are the characters in the string, where *n* is in the range 0 to 255. If *n* is odd (thereby making the total field length even), an extra zero pad byte is appended to ensure that the total field length is odd (if an OSTR field begins at an even byte offset, the following field will start at an odd offset, and vice-versa). This field can start and end at any byte offset, and is always at least 1 byte in length. This field type is included for compatibility with ResEdit; we strongly discourage using the OSTR field.

The Data Editor displays non-empty strings within double quotes.

PNT • Point

The PNT field (with a trailing space: 'PNT ') is 4 bytes long and encodes a QuickDraw `Point` structure. The high order 2 bytes are the Y coordinate; the low order 2 bytes are the X coordinate. The Data Editor displays the value of a PNT field as

(x,y) = (34,200) (or whatever the numbers are)

so that there is no ambiguity as to which coordinate is which. Each coordinate must be in the range -32768 to 32767. This field can start at any byte offset in the resource; however, even offsets are strongly recommended.

PPST • Even-Padded, Pad Included Pascal String

The PPST field is a variable length field containing a Pascal string. The first byte of the field contains the length, *n*, of the string, and the following *n* bytes are the characters in the string, where *n* is in the range 0 to 255. If *n* is even (thereby making the total field length odd), an extra zero pad byte is appended to ensure that the total field length is even (if an PPST field begins at an odd byte offset, the following field will also). If a null pad byte is necessary, the length field of the initial Pascal string is incremented to include the pad byte as well. This field can start and end at any byte offset, and is always at least 2 bytes in length.

The PPST field enables Resorcerer to parse portions of certain MPW resources. Its use is highly discouraged since you have to check both the first and the last byte of the field in order to tell exactly how many characters are in the string.

PSTR - Pascal String

The PSTR field is a variable-length field containing a Pascal string. The first byte of the field contains the length, *n*, of the string, and the following *n* bytes are the characters in the string, where *n* is in the range 0 to 255. Note that if there are an even number of characters in the string, the total field length will be odd (if the PSTR field begins on an even byte boundary, and has an odd number of bytes in it, the field after it will begin on an odd boundary). This field can start and end at any byte offset, and is always at least 1 byte in length.

The Data Editor displays non-empty strings within double quotes.

P*nmm* • Fixed Length Pascal String

The P*nmm* field is a fixed-length block of bytes in which to place a Pascal string, with the string's length byte placed at byte 0 of the block. The length of the block is taken to be $*nmm* bytes, where the first hex digit, *n*, is in the range '0' to '1', and the following 2 hex digits, *mm*, are each in the range '0' to 'F' (upper case only). This field can begin at any byte offset. The minimum size for this field is P001, since the empty Pascal string consists of a single length byte with value 0. The maximum length of any Pascal string is 255 bytes, which (along with its initial length byte) can always be stored in a P100 field. Bytes in the field beyond the logical end of the Pascal string are set to 0.

IMPORTANT COMPATIBILITY NOTE:

The P*nmm* field type is not compatible with ResEdit 2.1, which uses a similar scheme whereby *$nmm* encodes the maximum length of the string, not the length of the block of data in which the string is kept. Thus a P0FF field in ResEdit is the equivalent of a Resorcerer P100 field, and a P000 field in ResEdit is equivalent to a P001 field in Resorcerer. ResEdit is internally inconsistent on this front, since its fixed-length C string specification works the other way around (e.g. the same as Resorcerer).

The upshot of this is that if the Data Editor encounters a ResEdit style P*nmm* field it will allocate 1 less byte than ResEdit would.

Resorcerer features 7 field types for fixed-length blocks of string, text, hex, or other data. In all cases (that is, +*nmm*, -*nmm*, F*nmm*, P*nmm*, C*nmm*, T*nmm*, and H*nmm*, the hex value $*nmm* specifies the physical maximum size of the field, *regardless* of the type of data being kept in it.

Sorcery: If Resorcerer's Data Editor encounters a P0FF, P07F, P03F, P01F, P00F, P007, P003, or P001 field (corresponding to Pascal strings of length up to 255, 127, 63, 31, 15, 7, 3, and 1, respectively), it will ask you if the template is a ResEdit template, and if so whether you want to use it as is or stop what you're doing in order to fix it. These field types are the likeliest values to be found in a ResEdit template, and the least likely to be found in a Resorcerer template.

REAL • Single Precision 32-Bit Floating Point Number

The REAL field is 4 bytes long and encodes a single precision floating point number. In THINK C, these are declared as a `float`.

You can append a formatting command to the end of the REAL field's label string. The syntax of the command is similar to a standard C language `printf` floating point escape sequence: *%n*. *n* is the number of significant digits the Data Editor will use to display the floating point value; if the field label string does not exist, or is illegal, the Editor uses a default value.

RECT - Rectangle

The RECT field is 8 bytes long and encodes a QuickDraw `Rect` structure, which consists of four 2-byte coordinates in the order (top,left,bottom,right). The Data Editor displays the value of the RECT field as

(t,l,b,r) = (50,50,200,250) (or whatever the numbers are)

so that there is no ambiguity as to which coordinate is which. Each coordinate must be in the range -32768 to 32767. This field can start at any byte offset in the resource; however, even offsets are strongly recommended.

When you open this field for editing, a **Set** button lets you screen copy the coordinates of the rectangle. If the rectangular marquee is within the bounds of any other Resorcerer window, the coordinates are recorded into the field as local window coordinates; otherwise, they are taken as global screen coordinates.

Sorcery: Option-clicking on the field directly lets you set the rectangle bounds graphically.

RSID • Absolute or Relative Resource ID

The RSID field is 2 bytes long and contains a signed decimal absolute or relative resource ID in the range -32768 to 32767. RSID fields (as opposed to generic DWRD fields) enable the Data Editor to open the referenced resource for you while you are editing the Data resource. To determine the type of resource to open, the Editor scans the field's label from the end, looking for a four-character resource type enclosed in open and closed single quotes (e.g. 'STR ' or 'infs'), and uses the first one it finds. If it doesn't find a suitable match in the field label, it scans backwards in the resource for the first TNAM field that occurs at the same or previous nesting level, and uses its current value. Otherwise, the Editor can't determine the type and does nothing, in which case you have to use the standard resource opening/creation tools in your File Window.

When you open a RSID field to change the ID number, the Editor checks to see if a resource type is determinable. If one is, the editing dialog will contain an enabled **Edit** button that will either create the resource if it doesn't exist, or open it for editing if it does.

Relative resource IDs are numbers that must be added to a base resource ID before being used to reference a resource. The actual

base ID is kept in the label string. The Editor will interpret the value of the RSID field as a relative resource ID if the label string ends with the base decimal number followed by a plus sign, as in:

RSID Resource ID of extension scope info ('scop') -27136 +

Sorcery: Option-double-clicking on a RSID field in the Data Editor window will directly open (or create) the referenced resource if its type can be determined by the above-described scan.

SELF • Include This Template as a Counted List Item

The SELF field lets you design simple recursively structured resources (i.e. trees of similarly structured data). It must be the only field between the LSTC and LSTE that define a repeatable counted list item. The actual data for each node in your tree is described by the fields before and/or after the recursive counted list. The type of recursive ordering you get is dependent on whether the list occurs first, last, or in the middle in the template.

SFRC • SmallFract Fixed Point Number

The SFRC field is 2 bytes long and encodes a standard `SmallFract` number (as documented in *Inside Macintosh*, volume 5, chapter 8) in the range [0.0, 1.0). The SFRC field is the same as the fractional portion of a FIXD field. This field can start at any byte offset in the resource; however, even offsets are strongly recommended.

SKIP • Word Skip Offset

The SKIP field is 2 bytes long and contains an unsigned decimal long value between 2 and 65535, which represents the number of bytes, including the 2 bytes of the SKIP field itself, to skip to get to a matching SKPE field. Typically, intervening fields contain complex structures of variable size; skip offsets allow for fast scanning algorithms when your resource data has variable-sized items or fields in it that you need to skip over quickly. Although this field can occur at any byte offset in the resource, even offsets are strongly recommended.

SKIP...SKPE pairs may not be nested except as part of the standard nesting of repeated list or keyed items. SKIP fields are not editable, but are computed automatically for you. Their values are only shown when you have the Data Editor's **Show Offsets** option set.

SKPE • Skip End

The SKPE field allocates no data; it simply marks the position in the data that should be used to calculate where the previous matching BSKP, BSIZ, SKIP, WSIZ, LSKP, or LSIZ field should point to.

SKIP...SKPE pairs may not be nested except as part of the standard nesting of repeated list or keyed items. Skip offset fields are not editable, and their values are only shown in the Data Editor. When you have the **Show Offsets** option set, the Data Editor displays skips using pairs of small crop marks on the left. The first mark indicates the skip offset field and the last mark indicates the last field in the set of fields whose skip offset is being computed.

TNAM - Type Name

The TNAM field is 4 bytes long and encodes a four-character Macintosh system or resource type name. It is displayed surrounded by single quotes. This field can start at any byte offset in the resource; however, even offsets are strongly recommended. If any character value is less than $20, or equal to $7F, the Editor will display the entire TNAM value in hex. You can enter any character's value in this field's value-editing dialog either as four characters, or as a nine-character string starting with a '$' and followed by exactly eight hex digits (leading 0's required).

When new resource data is created, TNAM fields are set to '????'.

T*nmm* • Fixed Length Text

The T*nmm* field is a fixed length block of bytes in which to place pure ASCII text, beginning at byte 0 of the block. The length of the block is taken to be $*nmm* bytes, where the first hex digit, *n*, is in the range '0' to '9', and the following 2 hex digits, *mm*, are each in the range '0' to 'F' (upper case only). This field can begin at any byte offset. The minimum block size is T000—that is, the empty block—which allocates 0 bytes of storage. The maximum block size encodable using this scheme is T9FF, or 2559 (decimal) bytes. Bytes in the field beyond the logical end of any text you enter are set to 0.

TXTS • Sized Text Dump

The TXTS field consists of 0 or more bytes of data to be displayed as text. The TXTS field must explicitly or implicitly be followed by a

SKPE field, so that the parser can know when to stop parsing text and start parsing subsequent fields. The amount of text parsed is equal to the value of the previous matching skip offset (BSKP, WSKP, or LSKP) or sizeof (BSIZ, WSIZ, or LSIZ) in the resource, minus the lengths of any intermediate fields.

The TXTS field lets you do a local dump of a variable length of text data up to a given length recorded earlier in the resource. For example, if in the sequence "WSIZ, REAL, REAL, REAL, TXTS, SKPE" the value of the WSIZ field were 32, then the TXTS field would be 32-3*4 bytes of text, since REAL fields are 4 bytes long.

UBYT • Unsigned decimal byte

The UBYT field is 1 byte long and encodes an unsigned decimal number in the range 0 to 255. This field's value-editing dialog will accept either decimal or hex input preceded by a '$'.

ULNG • Unsigned decimal long

The ULNG field is 4 bytes long and encodes an unsigned decimal number in the range 0 to 4,294,967,295. The field can begin at any byte offset in the resource; however, even offsets are strongly recommended. This field's value-editing dialog will accept either decimal or hex input preceded by a '$'.

UNIV • Universal THINK 96-bit Floating Point

The UNIV field is 12 bytes long and encodes a 96-bit extended precision floating point number. The field can start at any byte offset in the resource; however, long or quad word offsets are strongly recommended. The UNIV field has the same format as the XT96 field, except that the exponent portion of the floating point number is duplicated and placed in the 16 bits of otherwise unused pad space of the extended number. This allows an 80-bit extended float to be embedded within the 96-bit Universal float. Neither of the 96-bit formats have any more numerical precision than the 80-bit format; the extra padding bytes are used for machine addressing efficiency. For more information on these numeric types, see Symantec's THINK C manual and the *Apple Numerics Manual, Second Edition*.

You can append a formatting command to the end of the field's label string. The syntax of the command is similar to a standard C language `printf` floating point escape sequence: *%n*. *n* is the

number of significant digits the Data Editor will use to display the floating point value; if the field label string does not exist, or is illegal, the Editor uses a default value.

UWRD • Unsigned decimal word

The UWRD field is 2 bytes long and encodes an unsigned decimal number in the range 0 to 65535. The field can begin at any byte offset in the resource; however, even offsets are strongly recommended. This field's value-editing dialog will accept either decimal or hex input preceded by a '$'.

WBIT • Word Bit
WB*nn* • Word Bit Field

The WBIT and WB*nn* fields occur in groups specifying individual bits or groups of bits within a single word of data (16 bits). The WB*nn* field declares a bit field *nn* bits wide (01 <= *nn* <= 16) anywhere within the word, as long as the field does not cross the word's boundary. As WBIT and WB*nn* fields are encountered in order, they specify bits beginning with the most significant bit of the word. When the Data Editor displays each bit or bit field, it labels the field with the proper bit number or numbers (inclusive). The value of a single bit is shown as either "On" if the bit is set, or "Off" if the bit is clear. The value of a bit field is numeric. You can change the value of single bits using the Data Editor's **Toggle Value** menu command. The group of 16 bits can begin at any byte offset in the resource; however, even offsets are strongly recommended.

WFLG • Word Flag

The WFLG field displays the value of the low-order bit 0 of a word. The other 15 bits are not editable and set to 0. You can change the flag value using the Data Editor's **Toggle Value** menu command.

WHEX • Word Hex Data

The WHEX field is a variable length field containing a 2-byte unsigned word followed by a block of data of unknown format. The first 2 bytes of the field contain the length, *n*, of the block, and the following *n* bytes are the pure hex data, where *n* is in the range 0 to 65,535. This field can start and end at any byte offset, and is always 2+*n* bytes in length. It is usually a good idea to follow WHEX fields with an AWRD or ALNG field. The initial length field is not displayed, but is automatically computed for you.

WSHX • Word Skip Hex Data

The WSHX field is a variable length field containing a 2-byte long followed by a block of hex data of unknown format. The first 2 bytes of the field contain the skip offset, *n+2*, of the block, and the following *n* bytes are the pure hex data, where *n* is in the range 0 to 65,533. This field can start and end at any byte offset, and is always 2+*n* bytes in length. It is usually a good idea to follow WHEX fields with an AWRD or ALNG field. The initial skip offset is not displayed, but is computed automatically for you.

WSIZ • Word Sizeof

The WSIZ field is 2 bytes long, and contains an unsigned decimal value between 0 and 65,535 that represents the number of bytes in the following data, up to a matching SKPE field. Typically, intervening fields contain complex structures of variable size or unknown format. The byte count stored in the WSIZ field does not include its own size.

WSIZ...SKPE pairs may not be nested except as part of the standard nesting of repeated list or keyed items. WSIZ fields are not editable (their values are automatically computed for you), and their values are only shown when you have the **Show Offsets** option set in the Data Editor.

We discourage the use of WSIZ fields in favor of SKIP/WSKP fields, which are more elegant and used more often in Apple resources.

WSKP • Word Skip Offset

The WSKP field is 2 bytes long, and is the same as a SKIP field. It is included for completeness. See the SKIP explanation for more on its interpretation.

WSTR - Word String

The WSTR field is a variable-length field containing a 2-byte word followed by a string of ASCII characters. The first word of the field contains the length, *n*, of the string, and the following *n* bytes are the characters in the string, where *n* is in the range 0 to 65535 (note that the Data Editor can only edit up to 32K bytes of text at one time). This field can start and end at any byte offset, and is always 2+*n* bytes in length. Note that if there are an odd number of characters

in the string, the total length will be odd (if the WSTR field begins on an even byte boundary, and has an odd number of bytes in it, the field after it will begin at an odd offset). It is usually a good idea to follow WSTR fields with an AWRD or ALNG alignment field.

The Data Editor displays non-empty strings within double quotes.

XT80 • Extended Precision 80-bit Floating Point Number

The XT80 field is 10 bytes long and encodes an 80-bit SANE extended precision number. Each of these fields can start at any byte offset in the resource; however, long word offsets are strongly recommended. This field is the same as the EXTN field.

XT96 • Extended Precision 96-bit Floating Point Number

The XT96 field is 12 bytes long and encodes a 96-bit extended precision number. Each of these fields can start at any byte offset in the resource; however, long or quad word offsets are strongly recommended for XT96 fields. XT96 field represents a floating point number used internally by Apple's SANE library and the 68881 FPU chip in some Macs. They have the same numerical precision as 80-bit floats; the difference in their lengths is due to extra zero pad bytes that make the total length of the float a multiple of the length of a long word, which in turn speeds certain kinds of hardware memory access. For more information, see the *Apple Numerics Manual, Second Edition*.

You can append a formatting command to the end of the field's label string. The syntax of the command is similar to a standard C language `printf` floating point escape sequence: *%n*. *n* is the number of significant digits the Data Editor will use to display the floating point value; if the field label string does not exist, or is illegal, the Editor uses a default value.

ZCNT • Zero-based Count

The ZCNT field is two bytes long and contains a signed integer, *n*, in the range from -1 to 32767. This number represents 1 less than the number of repeated items in the first counted list beginning with a LSTC field that occurs anywhere after the ZCNT field at the same item nesting level. As you add or delete items from the list, this field is automatically updated for you; it is otherwise uneditable.

This field type is included for compatibility with some of the early

Macintosh resource types, but its use is discouraged. Use the one-based counts (BCNT, OCNT, or LCNT) instead.

The label for this field should be the plural name of the items to be counted, such as “Networks”, or “Menus”. For best results, do not use labels that start out “Number of” or “# of”, such as “Number of networks” or “# of Menus”, since the Data Editor copies the value of this label as a header string for each item in the displayed list.

EXAMPLES OF TEMPLATES

This section shows you a variety of examples, both simple and complex, of how to use the various template fields to describe your custom resources. Many of the examples are related to each other, so you should (at least the first time) read them through consecutively.

For an example of how to build a filtered template, see the "Designing Filters" section later in this chapter.

It is also very instructive to look at some of the TMPL resources in the various files in the "Resorcerer® Templates" folder distributed with your copy of Resorcerer. These templates range in complexity from quite simple ('sysz' resources with one field) to extremely complicated (QuickDraw 'PICT' resources with over 3000 fields!).

EXAMPLE

Suppose you want to create a custom 'QUAD' resource, which consists of the integer coordinates of the four vertices of an arbitrary quadrilateral.

SOLUTION

QuickDraw uses signed 16-bit words to encode coordinate values. A minimal solution would be to create a template consisting of eight fields: the vertical and horizontal coordinates of each of the four quadrilateral vertices:

```
DWRD    X coordinate of first vertex
DWRD    Y coordinate of first vertex
DWRD    X coordinate of second vertex
DWRD    Y coordinate of second vertex
DWRD    X coordinate of third vertex
DWRD    Y coordinate of third vertex
DWRD    X coordinate of fourth vertex
DWRD    Y coordinate of fourth vertex
```

This describes a resource of four pairs of two-byte words, beginning with the X coordinate of the first vertex and ending with the Y coordinate of the fourth vertex.

EXAMPLE

QuickDraw uses the basic `Point` type to hold coordinate pairs, and you would like to read the resource data directly into a structure consisting of four `Point`s. The previous solution won't work, since `Point`s store their vertical coordinates first.

SOLUTION

Use the 'PNT ' field type (don't forget the final space character) for QuickDraw points.

```
PNT         First vertex
PNT         Second vertex
PNT         Third vertex
PNT         Fourth vertex
```

EXAMPLE

You decide that you want your 'QUAD' resource to be an array of quadrilaterals, so that you can keep any number of them in one resource rather than in lots of individual resources. Each quadrilateral takes up 16 bytes of storage, so the total number of quadrilaterals in the resource will be computed by taking the total size of the resource in bytes and dividing by 16.

SOLUTION

Use the List Begin (LSTB) and List End (LSTE) fields to indicate the beginning and end of a set of template fields that can be repeated indefinitely until the end of the resource data.

```
LSTB
      PNT         First vertex
      PNT         Second vertex
      PNT         Third vertex
      PNT         Fourth vertex
LSTE
```

The Data Editor will display each set of four 'PNT ' fields indented and marked as one list entry, and will allow you to create and delete these as single items.

EXAMPLE

Rather than calculate the number of quadrilaterals in the resource, as in the previous example, you would like an explicit count placed just prior to the repeated list in the resource data. If the list is empty, you want the initial count to be set to 0. We may want to add other data after the list, which the previous indefinitely repeated list would preclude being able to do.

SOLUTION

```
OCNT        Quadrilaterals
LSTC
      PNT           First vertex
      PNT           Second vertex
      PNT           Third vertex
      PNT           Fourth vertex
LSTE
```

The 1-based Count (OCNT) field takes up two bytes and is labeled with the name of the item that will be repeated, in this case “Quadrilaterals”. Instead of the List Begin (LSTB) field used in the last example, you have to use the Begin Counted List (LSTC) field to make sure that the Data Editor knows it should place the item count into the OCNT word prior to the first item of the list.

For instance, if the list consists of three quadrilaterals, the bytes 0-1 will contain the number 3, bytes 2-17 (decimal) will contain the first quadrilateral’s four vertices, bytes 18-33 will contain the second quadrilateral’s four vertices, and bytes 34-49 will contain the vertices of the third quadrilateral.

EXAMPLE

Another resource you might design would be a ‘POLY’ (or maybe ‘PGON’) resource, which will contain the data for an arbitrary polygon. This would be a list of vertices preceded by the number of vertices. However, you want to keep the coordinates in a virtual floating point system, as opposed to QuickDraw points. Furthermore, you want each polygon to be associated with a name, a color, and a set of 16 flag bits.

SOLUTION

```
HWRD        Flag bits
ESTR        Polygon name
OCNT        Vertices
COLR        Polygon fill color
LSTC
      REAL          X coordinate %6
      REAL          Y coordinate %6
LSTE
```

Notice that it is not necessary for the OCNT field to be directly next to its list of counted items. The Editor allows count fields to be placed anywhere prior to the start of the list, as long as it's at the same nesting level as the list and as long as there are no intervening counted lists to confuse things. In general, however, it is a good idea to start your counted lists just after the count field unless there is a good reason not to.

Since the ESTR field starts on an even offset into the resource data, the OCNT field will also. The offset at which the OCNT field data begins must be determined by looking at the length of the string kept in the 0'th byte of the ESTR field, and adding 1 to it (since you need to include the length byte itself). If that number is odd, add 1 again to account for the pad byte; the result is the number of bytes from the beginning of the ESTR field to the beginning of the OCNT field. When the Data Editor displays the coordinate values, it will use six significant digits for both X and Y.

EXAMPLE

In the previous example, the polygon resource's flag bits would only be editable as hex. Since individual bits have meaning and position within the word, you would like the Data Editor to help you remember which bit is which. Furthermore, you would like to pre-allocate a fixed amount of storage into which the polygon's name string will be stored, so that you know at exactly which offset the subsequent field will start. And while you're at it, put the count field back next to the list, and keep the coordinate values in a high-precision floating point system. You also want to plan ahead and reserve some space in the coordinate list for a third dimension.

SOLUTION

```
WBIT        Traverse clockwise (high-order bit in word)
WBIT        Fill when drawing
WBIT        Use color
WBIT        Self-intersecting
WB03        Outline thickness (0-7)
WB09        Reserved
C040        Polygon name
COLR        Polygon fill color
OCNT        Vertices
LSTC
      DOUB        X coordinate %10
      DOUB        Y coordinate %10
      FLNG        Reserved (in case we go to 3 dimensions)
      FLNG        Reserved (in case we need a Z double coordinate)
LSTE
```

The four WBIT and two WB*nn* fields encode each bit, from bit 15 down to bit 0, of the first word of the resource data. The upper four of these bits have been labeled with what they stand for, the next 3-bit field (WB03) contains a decimal number for the thickness of any border outline, and the rest of the bits have been reserved for enhancements or other properties. The Data Editor shows you the bit numbers when you edit them, and recognizes the "Reserved" label in order to make the bit field uneditable.

The C040 field allocates $040 (64 decimal) bytes in which you can place a C string for the null-terminated polygon name (maximum name size will be 63 characters). This way, the COLR field data is guaranteed to start at offset 66 (decimal) in the resource data. This lets you do the following (which you would not be able to do using the previous example with its variable length string field) in C:

```
typedef struct {
     short double x;   /* sizeof(short double) == 8 */
     short double y;
     short double unused_z;

} RealPoint;
```

```
typedef struct {
     unsigned short    clockwise : 1,
                       fill : 1,
                       useColor : 1,
                       isIntersecting : 1,
                       outline : 3,
                       reserved : 9;
     char name[64];
     RGBColor fillRGB;
     short numVertices;
     RealPoint firstVertex[];

} PolygonHeader;

PolygonHeader **poly;
RealPoint *vertex;

poly = (PolygonHeader **)GetResource('POLY’,128)
if (GoodResource(poly)) {
     HLock(poly);

     n = (**poly).numVertices;
     vertex = (**poly).firstVertex;

     for (k=0; k<n; k++,vertex++) {
           /* Process the next vertex */
           }

     HUnlock(poly);
     }
```

EXAMPLE

Each face of a polyhedron is a polygon, with any number of polygons making up the polyhedron. Since each polygon is itself a list of fields, you might want to design a 'HDRN' resource that is a list of lists. For now, you no longer care about naming each polygonal face of the polyhedron, but want a name for the polyhedron itself. Since you know that your application only deals with polyhedra with only a few faces, you can keep the face count in a byte instead of a word. Since the previous field is also a single byte, the beginning of the counted list of faces will be on an even offset. You also want to use true 3D coordinates this time.

SOLUTION

```
WBIT        Traverse clockwise (high-order bit in word)
WBIT        Fill when drawing
WBIT        Use color
WBIT        Self-intersecting
WB03        Outline thickness (0-7)
WB09        Reserved
P100        Polyhedron name
DBYT        Transformation index
BCNT        Faces
LSTC
    WB16        Polygon flag bits
    OCNT        Vertices
    LSTC
        DOUB        X value %10
        DOUB        Y value %10
        DOUB        Z value %10
    LSTE
LSTE
```

The P100 field pre-allocates the maximum of 256 bytes to keep a Pascal string for the polyhedron name. Following it is a single byte containing a number that you plan to use to index an array of transformations that your program supports. After this is a byte containing the number, *n*, of polygon faces following. Each of these *n* polygons consists of a word of flag bits (WB16), followed by a count, *m*, of polygon vertices, followed by *m* vertex items in the list.

EXAMPLE

The previous example contained a single byte labeled the transformation index, a number between 0 and N-1, where N is the number of transformations your application currently supports. But when editing the resource with the Data Editor, you would like help remembering the meanings of the index values that go in that field. You also want to add a symbolic constant for the outline thickness to help remind yourself that a thickness of 0 means no outline (that is, it's a special case). And while you're at it, you want to use an even higher precision floating point type and add some trailing fields after the doubly nested list.

SOLUTION

```
WBIT        Traverse clockwise (high-order bit in word)
WBIT        Fill when drawing
WBIT        Use color
WBIT        Self-intersecting
WB03        Outline thickness (0-7)
    CASE        No outline=0
WB09        Reserved
P100        Polyhedron name
DBYT        Transformation index
    CASE        90° Counterclockwise=1
    CASE        90° Clockwise=0
    CASE        Skew=2
    CASE        Mirror=3
    CASE        Dual=4
    CASE        Stellate=5
BCNT        Faces
LSTC
    WB16        Polygon flag bits
    OCNT        Vertices
    LSTC
        EXTN        X value %10
        EXTN        Y value %10
        EXTN        Z value %10
    LSTE
LSTE
DATE        Creation date
RSID        Picture ('PICT') resource ID
```

The CASE fields define symbolic names for the important values that

you might want to install in any given field. The characters in the label field for each CASE field up to the '=' sign are used to create a pop-up menu for you to choose from when you edit the field's value in the Data Editor. Whichever menu choice you make causes the value string to the right of the '=' sign in the above label string to be installed. The pop-up menu's entries will be created in the same order as the above CASE statements appear in the template.

For algorithmic reasons, our application needs to encode a clockwise transformation as 0 and counterclockwise as 1. But the more common transformation is the latter. So we place the counterclockwise CASE first in the list of symbolic values so that each time we create one of these resources the DBYT field will be set to 1 (counterclockwise) by default.

Each coordinate is now kept in an 80-bit extended double floating point field. And after the list of faces, you've added a date time stamp field, and a resource ID field for a related 'PICT' resource that illustrates this polyhedron. You can now open or create this 'PICT' resource directly by Option-double-clicking on the resource ID field when you edit it.

EXAMPLE

When scanning the polyhedron resource, there are times when you want to get directly to the resource ID field because you don't care about the intervening variable length lists. To skip across the lists, though, you need to know how far to go, something that is dependent on the lengths of the lists and their items. The length can be different for every resource.

SOLUTION

```
SKIP        Offset to get to illustration's resource ID
    WBIT        Traverse clockwise (high-order bit in word)
    WBIT        Fill when drawing
    WBIT        Use color
    WBIT        Self-intersecting
    WB03        Outline thickness (0-7)
        CASE        No outline=0
    WB09        Reserved
    P100        Polyhedron name
    DBYT        Transformation index
        CASE        90° Counterclockwise=1
        CASE        90° Clockwise=0
        CASE        Skew=2
        CASE        Mirror=3
        CASE        Dual=4
        CASE        Stellate=5
    BCNT    Faces
    LSTC
        WB16        Polygon flag bits
        OCNT        Vertices
        LSTC
            EXTN        X value %10
            EXTN        Y value %10
            EXTN        Z value %10
        LSTE
    LSTE
    DATE        Creation date
SKPE
RSID        Picture ('PICT') resource ID
```

The SKIP field here contains how many bytes to add to the SKIP field's address (corresponding to offset 0 in this particular resource) in order to get the address of the resource ID word.

EXAMPLE

Suppose you wanted to reference all your 'HDRN' polyhedron resources from a master table of contents in a 'HDRA' resource. There are two ways to reference another resource of a given type, either by specifying its resource ID or its resource name. Therefore, there are two alternate structures your table of contents might take on. The first uses resource names, and the second method uses resource IDs. Furthermore, you want to be able to override the default resource type for any named resource by explicitly specifying another possible type, unless that type is 0, in which case the default should be used.

SOLUTION

```
TNAM        Default polyhedra resource type (usually 'HDRN')
KWRD        Format of table
      CASE          Use resource names=0
      CASE          Use resource IDs=1
KEYB        0
      OCNT          Polyhedra resources
      LSTC
            TNAM          Override polyhedron resource type
                  CASE          Use default=$00000000
            ESTR          Resource name
      LSTE
KEYE
KEYB        1
      OCNT          Polyhedra resources
      LSTC
            RSID          Resource ID
      LSTE
KEYE
```

In this solution, you make use of a key field and its keyed item list. If the KWRD key value is 0, then the following resource data will have the structure of a counted list of override resource types and their resource names. If the key is 1, then the following resource data will have the structure of a counted list of resource ID words.

Note that Option-double-clicking on the RSID field in the Data Editor will open the resource whose type is given in the very first TNAM field, and not the TNAM field within the first keyed item. The RSID field is within the context of the first TNAM field but not within that of the second TNAM.

EXAMPLE

The previous example is not really general enough. You might want some resources to be referenced by ID while others are to be referenced by name, all within the same table of contents.

SOLUTION

```
OCNT        Polyhedra resources
LSTC
      TNAM          Polyhedra resource type
            CASE          Our standard=HDRN
            CASE          Apple standard=hdrn
      KWRD          Type of reference
            CASE          Resource ID=0
            CASE          Resource name=1
      KEYB          0
            RSID          Polyhedron resource ID
      KEYE
      KEYB          1
            ESTR          Resource name
      KEYE
LSTE
```

In the previous example, you had a variable keyed format, one keyed item consisting of one kind of list, and the other keyed item consisting of another type of list. In this solution, there is one list of items, each item of which has a variable keyed format. Thus, when scanning the list, the value of each key determines whether the next two bytes should be taken as a resource ID, or whether the next variable number of bytes should be taken as a Pascal string.

You use an even-padded Pascal string to align on a word boundary at the end of every list item. This guarantees that the start of the next list item (the TNAM field) is always on an even boundary.

To make it easier to set the TNAM field for each item in the list, you add a CASE menu to the field for the two most common cases of polyhedra resource types.

EXAMPLE

Suppose that someone else's application implements a third type of resource referencing, using a key value of 2 to indicate that subsequent data is in a new format (for instance, file name, directory ID, resource type and resource index). The problem is that you don't know what the format is, because you only have the old template. The Data Editor will be unable to open the new resource if any item in the list of variant keyed items makes use of the new format. This is a general format extensibility problem that frequently arises, and it is a problem because it keeps you from editing later fields in the data whose formats your template does know about.

SOLUTION

The solution is to think ahead and design in a skip field around your keyed items before you let the rest of the world begin creating them also. You need to add the unknown keyed item to take care of holding the data of those items in the third (or other) format that your template does not yet know about. Apple's Balloon Help 'hmnu' resource is a good example of this style of variant item.

```
OCNT        Polyhedra resources
LSCT
      TNAM          Polyhedra resource type
            CASE          Standard=HDRN
            CASE          AppleStd=hdrn
      WSKP          Offset to next item
            KWRD          Type of reference
                  CASE          Resource ID=0
                  CASE          Resource name=1
            KEYB          0
                  RSID          Polyhedron resource ID
            KEYE
            KEYB          1
                  ESTR          Resource name
            KEYE
            KEYB
                  HEXD          Unknown reference format
            KEYE
      SKPE
LSTE
```

EXAMPLE

Suppose you want to design an 8 by 16 cell array, in which various symbols will appear. The symbols are taken from a special symbol font and so are indexed by a unique ASCII character in the range 0 - 127. You want to keep the array in an 'ASYM' resource.

SOLUTION

T010	Bytes 0 - 15
T010	Bytes 16 - 31
T010	Bytes 32 - 47
T010	Bytes 48 - 63
T010	Bytes 64 - 79
T010	Bytes 80 - 95
T010	Bytes 96 - 111
T010	Bytes 112 - 127

With the above fields, the Data Editor will display each consecutive 16 bytes of the resource in a single field of type T010.

EXAMPLE

The first 32 values of the above array represent generally non-printable characters that may not be easily viewable or even editable when you open a T010 field for editing. Furthermore, ASCII 127 (the Delete character) is also better displayed in hex.

SOLUTION

H010	Bytes 0 - 15
H010	Bytes 16 - 31
T010	Bytes 32 - 47
T010	Bytes 48 - 63
T010	Bytes 64 - 79
T010	Bytes 80 - 95
T010	Bytes 96 - 111
T00F	Bytes 112 - 126
H001	Byte 127 (DEL)

The first two and the last rows of the array, representing these non-printable characters, are made editable in hexadecimal, rather than attempting to print them as text.

EXAMPLE

Suppose you want to keep a resource that contains a family tree of names and ages, along with a list of all children for each name. In this case, we can use a counted recursive list.

SOLUTION

```
ESTR        Name
DWRD        Age
OCNT        Children
LSTC
      SELF
LSTE
```

This resource consists of a tree of families, each consisting of an even padded Pascal name string, and an age field, followed by a count of children, each of which is a node of the same structure.

A C routine to scan this resource data (assumed locked) might look something like this:

```
/*
 *    Scan a family tree whose parent name begins at
 *    data, which must be an even address.  Return
 *    the number of bytes scanned.
 */

long ScanFamily(register unsigned char *data) {
      short age, kids, len;
      unsigned char *name;

      name = data;           /* Start of family name */
      len = 1 + *data;              /* Size of name field */
      if (isOdd(len)) len++; /* Skip any pad byte */
      data += len;           /* Get to age field */
      age = *(short *)data;  /* OK, cause it's even */
      data += sizeof(short); /* Move on to count */
      kids = *(short *)data; /* Still at even address */
      data += sizeof(short); /* Get to start of list */

      /* Do something here with name, age, and kids */
      /* And then recursively scan all the kids */

      while (kids-- > 0)     /* For all children... */
          data += ScanFamily(data);   /* Do it again */

      return ( (long)(data - name) );
}
```

EXAMPLE

Using the last template for a family tree resource, you've built one or more extensive tree resources. But you've decided that it would be nice to record, in a new field, the sex of each person. The problem is that if you change the TMPL you won't be able to open the existing resources, whose old structure is no longer described by the new template.

SOLUTION

First, you must create an intermedate template. Use one of the insertion fields (in this case to insert a word) in the spot in the template where you want the new gender field to be recorded:

```
ESTR        Name
DWRD        Age
+WRD        Insert two bytes here
OCNT        Children
LSTC
      SELF
LSTE
```

Now open every tree resource. The newly inserted +WRD fields will be inserted recursively, set to zero, and marked dirty. Now, close and save the tree resource.

You can only open and close the resource once using a template with any insertion (or deletion) field. After the change, you must replace the insertion field with a properly labeled field, such as:

```
ESTR        Name
DWRD        Age
FBYT        Filler
DBYT        Gender
      CASE  Female=0
      CASE  Male=1
OCNT        Children
LSTC
      SELF
LSTE
```

Don't forget to change your parsing code too!

DESIGNING FILTERS

When a template ('TMPL') resource purportedly describes a given resource type, the first four characters of the template's resource name declare the type of data the template describes. However, if the first field in the template is of type 'FLTR', then the template declares to the Data Editor that the Editor should call upon a *filter* to pre-process the resource data into a temporarily different structure. The other fields in the template describe this temporary structure. When you finish editing the temporary structure, the Data Editor will call upon the filter a second time in order to convert the temporary structure back into the legal structure for the resource type. Filters let you edit indexed, encrypted, and/or compacted resources, among other things.

Filters are kept as code resources of type 'FLTR' containing a single subroutine. The subroutine can be compiled in either C or Pascal; the first character of the code resource's name should be either a 'C' or a 'P' to tell Resorcerer the type of calling sequence to use when it calls the filter.

The purpose of the filter is basically to take two handles of data, one the source and one the destination, and to convert the data in the source handle into a different form in the destination handle. Filter's used by the Data Editor should generally be able to perform the reverse transformation.

Any given filter can be responsible for filtering more than one resource type. The first four characters in the 'FLTR's resource name are used as flags, with the first character indicating the language. Currently, the second, third, and fourth characters should be set to spaces. The remaining characters in the name declare a list of the resource types (separated by a single space) that the filter pre- and post-processes. For example,

```
        FLTR  128    "C  SPUD HUNK"
```

is a filter code resource, whose main entry point will be called as a C subroutine every time the Data Editor is presented with

```
        TMPL  741    "SPUD"
or      TMPL  742    "HUNK"
```

in order to edit a resource of type 'SPUD' or 'HUNK' (but only if the template's first field is of type 'FLTR').

The resource IDs for both templates and filters can be arbitrary.

THE FILTER INTERFACE

Each filter code resource has one main entry point, called as a C subroutine or as a Pascal procedure, depending on the first character of the 'FLTR' resource name:

```
     void main(FilterRecord *fPtr);
or   PROCEDURE ResourceFilter(fPtr : FilterRecordPtr);
```

The `FilterRecord` structure is defined in the file "FLTR.h":

```
/* Types of filtering (messages) */

#define filterDeclareVersion    0
#define filterConvert           1
#define filterUnconvert         2
#define FILTERROOM 15

// Parallel vectors of C or Pascal callbacks to Resorcerer

typedef struct {
     void         (*PleaseWait)(void);
     Handle       (*GetResData)(ResType type, short id);
     } C_FltrCallbacks;

typedef struct {
     pascal void  (*PleaseWait)(void);
     pascal Handle(*GetResData)(ResType type, short id);
     } P_FltrCallbacks;

// FLTR's take pointers to a single FilterRecord as an
argument

typedef struct {
     long version;              // Version of Filter
     long message;              // Type of filtering to do
     long errCode;              // Error return code
     long sameOffsets : 1,            // TRUE when no data
moves
          reserved : 31;        // More flags to come

     Handle  original;          // "Resource" data
     ResType oType;                        // Data type
     short   oID;                    // Resource ID
     short   oAttrs;                 // Resource attributes

     Handle  converted;         // Filtered data
     ResType cType;                        // Data type
     short   cID;                    // Resource ID
     short   cAttrs;                 // Resource attributes

     union {                         // Resorcerer callback
functions
            C_FltrCallbacks C;      // C style
            P_FltrCallbacks P;      // Pascal style
            } call;
```

```
    long expansion[FILTERROOM];      // Room to grow
(zeroed)

    } FilterRecord, *FilterRecordPtr, **FilterRecordHandle;
```

The `version` field is for the filter code resource to declare itself to the caller when the `message` field is `filterDeclareVersion`. The only known version at this time is 0.

`message` contains a selector indicating what the caller is requesting the filter to do. Currently, only three selectors are defined:

```
filterDeclareVersion
filterConvert
filterUnconvert
```

`errCode` arrives preset to 0, and can be ignored if there are no errors. Otherwise, the filter should set it to a non-zero error code.

`sameOffsets` should be set only if the filter leaves the positions of every field in the input the same as in the output. If this bit is set, the Data Editor will not put up the usual alert when the user chooses **Show Offsets**.

Two sets of variables deliver resource information and data, one for the original input resource and one for the converted output. Both handles will be already allocated. If `message` is `filterConvert`, then `original` is the source and `converted` is the destination; otherwise, if `message` is `filterUnconvert`, then `original` is the destination and `converted` is the source. The resource type, ID, and attributes information is for completeness; most of it is usually unnecessary. However, if the 'FLTR' is reponsible for more than one type of resource, then you need to attend to the `oType` field to see what type of resource the caller is presenting to it.

Note: The data of the destination handle should be changed, *not* the handle itself, which arrives pre-allocated.

`call.C.PleaseWait` (`call.P.PleaseWait`) is a pointer to a C (Pascal) subroutine that animates the caller's "busy" cursor. The routine takes no arguments and should only be called if the filter finds itself in a long loop. When the filter's resource name starts with 'C', `call.C.PleaseWait` is a pointer to a C subroutine; if the name starts with 'P', then `call.P.PleaseWait` points to a Pascal procedure.

`call.C.GetResData(ResType type, short ID)` is a pointer to a C

function that returns a handle to a detached copy of a resource of a given `type` and `ID`, or `NIL` if none. As above, the language the function is written in is determined by the first character in the filter's resource name, and you should call the C variant or the Pascal variant accordingly.

Note: It is the filter's responsibility to dispose of any handle delivered to it by `GetResData` (using `DisposeHandle`).

`expansion` is an array of storage reserved for expansion for other callback routines or data fields that may be useful in the future. All unused slots are set to 0 (NIL).

FILTER EXAMPLE: INDEXED C STRING('CST#') RESOURCES

Standard string lists on the Mac are kept in resources of type 'STR#', which are described by the template:

```
OCNT        Strings
LSTC
      PSTR
LSTE
```

There are some constraining factors to this structure. The first is that no string can be longer than 255 characters, and the second is that the resource must be scanned sequentially in order to find the *i*'th string. The Toolbox's `GetIndString()` does just this. A third constraint might be that the number of strings can't be more than will fit in a 16-bit word (the OCNT field).

A more general and powerful resource would be something like this:

```
LCNT        Entries
LSTC
      HLNG          Offset to i'th string from resource start
LSTE
HLNG                Sentinel offset to first byte after last string
LSTB
      CSTR          String
LSTE
```

The difficulty with this structure is that the Data Editor cannot fill in the values of each HLNG offset field in the first list, since there is no

convention that links each entry to the string it points to in another arbitrary list. Another more subtle problem is that there is no way the Data Editor can guarantee that the two lists have the same number of items in them, which should be the case. In fact, there is nothing to say that the index entries are even in the same order as the strings.

Even though it would be possible to define yet more field types to take care of these particular situations, there will always be exceptions. Some resource types, for example, have initial indexes that contain more entries than the second list to which they refer, due to the need for a sentinel value at the end. Arbitrary conventions make it difficult to provide a general solution within the template description, particularly for resources that have already been designed as Rez source code to be compiled but not edited.

The solution is to define a filtered template for 'CST#' resources that describes the list of strings only, so that you never have to see or worry about the initial list of indexes. The associated filter simply deletes the initial index in the resource data, and presents the Data Editor with a structure described by the filtered template:

```
FLTR   These fields will be post-processed into the 'CST#' form
LCNT   Strings
LSTC
       CSTR   String
LSTE
```

When the Data Editor sees the FLTR field in the 'TMPL' whose name begins with "CST#", it looks for an associated filter ('FLTR') resource

whose name contains 'CST#' in the list of resource types it filters.

The C code for this filter might look like this:

```
#include "FLTR.h"

/* Prototypes for local routines */

void  DeleteIndex(FilterRecord *f);
void  AddIndex(FilterRecord *f);
int   HAppend(Handle dst, Handle src, long start, long size);
long  strlen(char *p);

void main(FilterRecord *f);

void main(FilterRecord *f) {    // Main entry point for
filter

      switch(f->message) {
            case filterDeclareVersion:
                  f->version = 0L;
                  break;

            case filterConvert:
                  DeleteIndex(f);
                  break;

            case filterUnconvert:
                  AddIndex(f);
                  break;
            }
      }

//    Delete the initial list of index offsets,
//    but leave long count field.

static void DeleteIndex(FilterRecord *f) {

      long numStrings,startIndex,startStrings;
      long endStrings,sizeStrings;

      /* Get index and strings from 'CST#' data */

      numStrings = *(long *)(*f->original);
      startIndex = sizeof(long);
      startStrings = startIndex + (numStrings+1) *
sizeof(long);
      endStrings = GetHandleSize(f->original);
      sizeStrings = endStrings - startStrings;

      /* Start with long count, add strings */
```

```
	SetHandleSize(f->converted,sizeof(long));
	*(long *)(*f->converted) = numStrings;

	f->errCode = HAppend(f->converted,
                             f-
>original,startStrings,sizeStrings);
	};

//	When unconverting, we build the index in original,
//	and then append the non-indexed list of strings.

static void AddIndex(register FilterRecord *f) {

	long size,numStrings,startIndex,sizeIndex;
	long startStrings,endStrings,sizeStrings;
	register long *offset,i,stringOffset;
	register char *p;

	numStrings = *(long *)(*f->converted);
	startStrings = sizeof(long);
	endStrings = GetHandleSize(f->converted);
	sizeStrings = endStrings - startStrings;

	// Allocate initial index

	startIndex = sizeof(long);
	sizeIndex = startIndex + (numStrings+1) * sizeof(long);
	SetHandleSize(f->original,sizeIndex);
	if (f->errCode = MemError())
		return;

	/* Scan packed C strings, and set index offsets */

	HLock(f->converted);
	HLock(f->original);

	// Install string and index count

	*(long *)(*f->original) = numStrings;

	// Prepare loop that includes extra sentinel offset at
end

	p = (*f->converted) + startStrings;
	offset = (long *) ((*f->original) + startIndex);
	stringOffset = sizeIndex;

	// For each string, set index entry...

	for (i=0; i<=numStrings; i++) {
		*offset = stringOffset;	/* Install index entry
*/
		offset++;			/* On to next entry */
		size = 1 + strlen(p);		/* Include null
byte */
		p += size;			/* On to next string */
		stringOffset += size;		/* And get its
offset */

		/* Keep user informed of long loop */

		if (i > 256) (*f->call.C.PleaseWait)();
		}
```

```
      HUnlock(f->converted);
      HUnlock(f->original);

      /* Now append strings after index */

      startStrings = sizeof(long);
      sizeStrings = GetHandleSize(f->converted) -
startStrings;
      f->errCode = HAppend(f->original,
                           f-
>converted,startStrings,sizeStrings);
      };

//    Append amount bytes of data from src to dst.
//    Deliver memory error or not.  Handles can be unlocked.

static int HAppend(Handle dst, Handle src, long start,long
amount)
      {
            long size; int err;

            size = GetHandleSize(dst);
            SetHandleSize(dst,size+amount);
            if (err = MemError()) return(err);

            BlockMove((*src)+start, (*dst)+size, amount);
            return(noErr);
      }

static long strlen(char *p)     // Private C string length
routine
      {
            char *str = p+1;

            while (*p++) ;
            return( (long)(p-str));
      }
```

FILTER VARIATIONS

Often, a set of related filtered resource types have related structures. In this case, you will find that converting each resource into and out of its temporary format uses the same or similar algorithms.

For instance, in the previous example of a filter for the indexed C string list ('CST#') resource, you might want also to define an indexed Pascal string ('PST#') resource. By including both resource types in the list of types in the 'FLTR's resource name, the filter can be responsible for converting both. Only minor changes are needed to accomodate both algorithms. Which resource type it should operate on is delivered to it in the `FilterRecord`'s `oType` or `cType` field for the data, as in:

```
      /* For each string, set index entry... */

      for (i=0; i<=numStrings; i++) {
            *offset = stringOffset;    /* Install index entry
*/
            offset++;                  /* On to next entry */
```