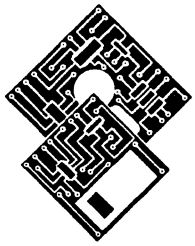

APPLE
PROGRAMMER'S
AND DEVELOPER'S
ASSOCIATION

290 SW 43rd. Street
Renton, WA 98055
206-251-6548

Macintosh Development Utilities

Version 1.0

APDA#: KMSDU1

About the Macintosh Development Utilities

Contents: Section:

Note: The catalog description of this product incorrectly states that there are three disks. That estimation was higher than necessary, and all of the files have been put on two disks.

The Macintosh Plus SCSI Developer Information chapter has been included in a new APDA product, SCSI Development Package.

Disks Included in the Macintosh Development Utilities

Macintosh Utilities 1:

```
Edit
Resource Editors/Creators folder:
     Dialog Creator
     Dialog Creator Example
     REdit
     ResEdit
     RMaker
```

Macintosh Utilities 2:

```
AppleTalk folder:
     Peek 3.0
     Poke
     Poke Packets
Graf3d Stuff folder:
     Graf3D.Rel
     Graf3DEqu.txt
PrintCalls.rel
Tools / Applications folder:
     DivJoin 1.0d8
     FreeTerm 1.8
     Localizer
     MacsBug
     Menu Capture
     PS Dump 2.5
     Screen Maker
     JumpStart Folder:
          JumpStart
          JumpStart Log
     Trap Timer folder:
          Scrapbook File
          Trap Timer DA
          Trap Timer Glossary
```

Description of the Contents of the Disks

- `Edit` is version 2.0.1b1 of Apple's original text-file editor.
- `Dialog Creator` is a specialized resource editor for creating dialogs. See section 3.
- `REdit` is a resource editor that is especially useful for localizing applications in other languages. See section 2.
- `ResEdit` is version 1.0 of Apple's resource editor. See section 1.
- `RMaker` is the original resource creator. It creates resources from descriptions in text files. See section 4.
- `Peek` allows "peeking" into an AppleTalk network to monitor packets as they are transmitted. See section 12.
- `Poke` allows "poking" packets onto the AppleTalk network. See section 13.
- The files in the `Graf3D Stuff folder` are necessary for development of applications that access the 3-D Graphics routines.
- `PrintCalls.rel` is a replacement for the same-named file in the Macintosh 68000 Development System, version 2.0.
- `DivJoin` is the newest version 1.0D9 of this program. It is used to break up large files for storage on 400K or 800K disks. See section 9.
- `FreeTerm` is a terminal emulator. See section 10.
- `Localizer` is used to adapt the Macintosh keyboard to match the local style.
- `MacsBug` is the Macintosh debugger, version 5.2.
- The `Menu Capture` installer document is used to allow capturing of the screen while the mouse is pressed in a menu. See section 8.
- PS Dump will transmit a text file of PostScript commands over AppleTalk to a LaserWriter. This is version 2.5 of this program.
- `Screen Maker` is a program to convert MacPaint picture files to picture files that can be used as startup screens.
- The `JumpStart folder` contains the `JumpStart` desk-accessory and the `JumpStart Log` application, which can be used to speed up loading of resources at application launch. See section 5.
- The `Trap Timer folder` contains the `Trap Timer` desk-accessory which can be used to monitor time spent while accessing traps. The `Scrapbook file` and `Trap Timer Glossary` can be used to view the output in a formatted report.

Documents in the Macintosh Development Utilities Package

This document, **About the Macintosh Development Utilities,** describes the contents of this package, i.e. the disks and the documentation.

ResEdit: A Macintosh Resource Editor describes ResEdit, a full featured utility used to edit resources, allowing the user to modify old resources or to create new ones.

REdit: A Macintosh Resource Editor describes REdit, a resource editor which is ideal for changing applications from one language to another. It does not support editing all resource types.

For creating dialogs easily, **Dialog Creator Instructions** tell how to use this useful utility.

RMaker is the resource creator which uses a text file description of resources as input for creating resources. This document describes how to use this application.

The **JumpStart Documentation** describes how to use JumpStart to super-charge applications and make them load faster. Note the cautions in this document that not all applications will work after super-charging.

Debugging with MacsBug describes the Macintosh Debugger MacsBug. All commands are detailed here. Also included is a quick reference page which can be removed.

Trap Timer Desk-Accessory Documentation tells about the Trap Timer, which can be used to discover where an application is spending its time.

The **Menu Capture Patch** allows capturing a screen while a menu is extended on Macintosh Plus computers. Note the cautions in this section about redistribution and use.

Instructions for **DivJoin 1.0d8** are included here. DivJoin allows dividing a file that is larger than 400k (or 800k for double-sided drive users) into sets of files that can be stored on floppies to be joined back together at a later time.

The **FreeTerm User's Manual** tells how to use this public-domain terminal program.

AppleTalk Information, AppleTalk Peek, and AppleTalk Poke describe how to program with AppleTalk, and how to monitor or sent packets over the AppleTalk network.

To speed up printing jobs on an ImageWriter, the document **Some Words of Wisdom About Using QuickDraw while Printing** tells some steps that can be taken to save printing time.

The March 1985 ImageWriter: Programmer's Notes gives some helpful tips for printing to the ImageWriter: calls available, how to change paper sizes, and more.

Optimizing Code for the LaserWriter gives some ideas for speeding up LaserWriter printing.

The next four documents are for reference only and are not up to date. They are accurate for the 64K ROM, but not for the 128K ROM. The first two, **Low Memory in Alphabetical Order** and **Low Memory in Numerical Order** list low memory globals. The **Trap List** is just that, a list of the Macintosh traps that are in ROM. The **Commented Call List** gives a good idea what traps and routines are called as a result of calling a specific trap.

The **Registration of File Types and Application Signatures** page is for sending to Macintosh Technical Support so that no two applications have the same signatures assigned to them.

A new version of the **Power User's Short-Cut Summary** document is included. These pages tell of short-cuts that can be used to increase productivity in your use of the Macintosh. Many of these tips are in manuals, but are summarized here for quick reference.

Version Numbers describes Apple's numbering scheme for labeling versions of an application.

The **SANE Information** document tells of requirements for using SANE. If you plan to use SANE, make sure that you read this document.

Welcome to Maug™ describes the Micronetworked Apple Users Group on CompuServe. This is a good place to communicate with other Apple product developers and to find utilities.

ResEdit:
A Macintosh Resource Editor

This document describes the 1.0A1 release of ResEdit.

Contents:

Note: As in *Inside Macintosh*, resource type names are shown within single quotes; for example, 'STR '. The quotes are not part of the name.

About ResEdit

ResEdit is an interactive, graphically based tool for manipulating the various resources in a Macintosh application. It lets you create and edit all resource types except 'CODE', and to copy and paste all resource types (including 'CODE'). ResEdit actually includes a number of individual resource editors: these include a **general resource editor**, for editing any resource in hex format, and several individual resource editors for specific types of resources. You can also write your own resource editors to use with ResEdit.

ResEdit is especially useful for creating and changing graphic resources such as dialogs or icons. For example, you can use ResEdit to put together a quick prototype of a user interface and try out different formats and presentations of resources. ResEdit is also useful for translating resources into a foreign language without having to recompile the program.

You can also extend ResEdit by creating templates for your own resource types. The generic way of editing a resource is to fill in the fields of a dialog box—this is the way you currently edit 'BNDL's, 'FREF's, 'APPL's, etc. The layout of these dialog boxes is determined from a template in ResEdit's resource file—you can add templates to edit new resource types.

Using ResEdit

To start ResEdit from the Finder, select and open the ResEdit icon. ResEdit displays a window that lists the files for each disk volume currently mounted.

Working with Files

To list the resource types for a file, select and open the file name from the list. (You can select a name by clicking it or by typing one or more characters of the name.)

When a disk volume window is the active window, the File menu commands act as follows:

New	Creates a new file.
Open	Opens the selected file (this is the same as double-clicking on the file name).
Get Info	Displays file information and allows you to change it.
Close	Closes the volume window (this is the same as clicking the close box). If it's a 3 1/2-inch disk, the disk will be ejected.

Caution: You can edit any file shown in the window, including the System file and ResEdit itself. However, it's dangerous to edit a file that's currently running. Edit a copy of the file instead (for example, the System file on a non-boot volume).

Note that ResEdit will recognize a new disk when it's inserted and also handles more than one drive.

Note also that you can use ResEdit to copy or delete files.

Working within a File

When you open a file, a window displays a list of all the resource types in that file. While this window is the active window, you can create new resources, copy or delete existing resources, and paste resources from other files.

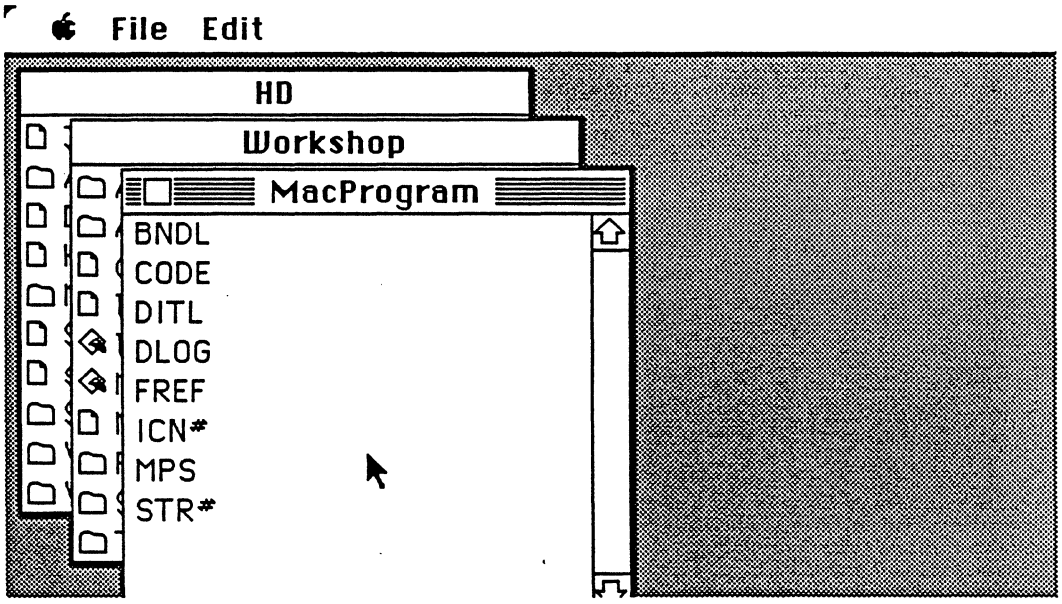

Figure 2 A File Window

When a file window is the active window, the **File menu** commands have the following effects:

New	Creates a new resource in the open file.
Open	Opens a window displaying all resources of the resource type selected (select the resource type by clicking it or by typing its first character). Note: The resources are displayed by a resource picker—the general resource picker displays the resources by type, name, and ID number; there are also special resource pickers for some resource types (for example, the 'ICON' resource picker displays the icons graphically). If you hold down the Option key while opening, the resource window will open with the general resource picker.
Open General	Opens the general resource picker.
Close	Closes the file window and asks if you want to save the changes you made. **Never reboot before closing!** If you have made any changes, rebooting before closing all file windows can leave the resource files in an inconsistent state.
Revert	Changes the resource file back to the version that was last saved to disk.

The **Edit menu** commands have the following effects:

Cut	Removes all resources of the resource types selected, placing them in the ResEdit scrap.

Copy	Copies all resources of the resource types selected into the ResEdit scrap.
Paste	Copies the resources from the ResEdit scrap into the file window's resource type list.
Clear	Removes all resources of the resource type selected, without placing them in the ResEdit scrap.
Duplicate	Creates duplicates of all resources of the resource types selected, and assigns a unique resource ID number to each new resource.

Working within a Resource Type

Opening a resource type produces a window that lists each resource of that type in the file. (This list will take different forms, depending on the underlying resource picker that's invoked; if you hold down the Option key during the open, the general resource picker is invoked.)

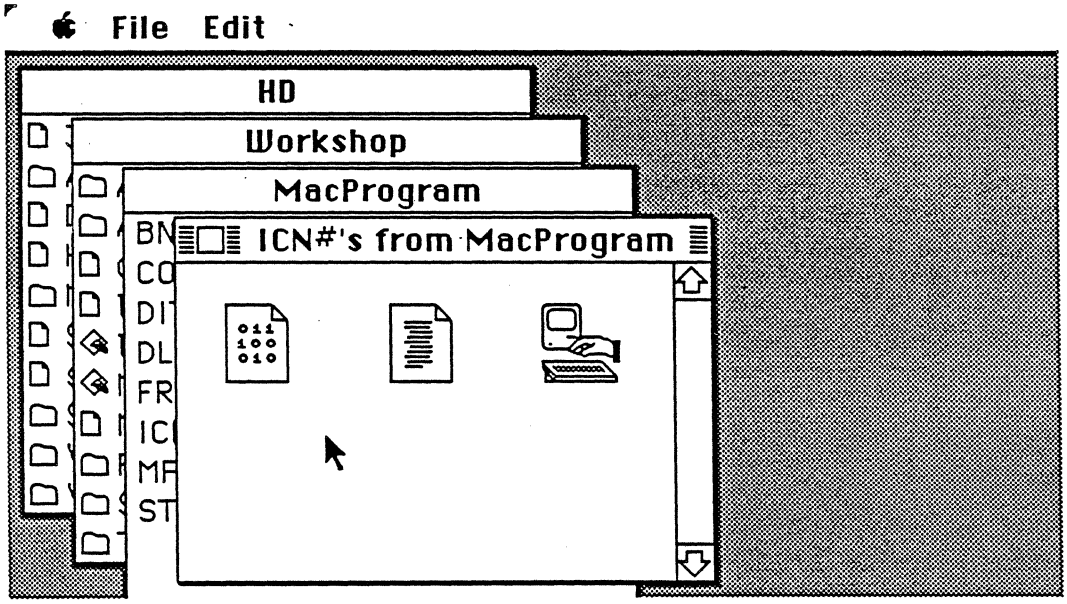

Figure 3. A Resource Type Window

When a resource type window is the active window, the **File menu** commands have the following effects:

New	Creates a new resource and opens its editor. A selection window is presented to allow you to select the resource type to create.
Open	Opens the appropriate editor for the resource you selected.
Open General	Opens the general (hexadecimal) resource editor.

Close	Closes the resource type window.
Revert	Changes the entire file back to what it was before opening the resource type window.

The **Edit menu** commands have the following effects:

Undo	May or may not be selectable, depending on the specific editor in use.
Cut	Removes the resources that are selected, placing them in the ResEdit scrap.
Copy	Copies all the resources that are selected into the ResEdit scrap.
Paste	Copies the resources from ResEdit scrap into the resource type window's resource list.
Clear	Removes the resources that are selected, without placing them in the ResEdit scrap.
Duplicate	Creates a duplicate of the selected resources and assigns a unique resource ID number to each new resource.

Changing a Resource's Type

If you hold down the Option key when copying or duplicating a resource, a dialog box with a list of resource types will be presented. You may choose any of these types, or type in your own type name that the copy/duplicate will be forced to become—that is, you may coerce the copy to a new resource type. (BE CAREFUL!!! —know what you are doing before trying this.)

Editing Individual Resources

To open an editor for a particular resource, either double-click on the resource or select it and choose Open from the File menu. One or more auxiliary menus may appear, depending on the type of resource being edited. Some editors, such as the 'DITL' editor, allow you to open additional editors for the elements within the resource. All the editors use File and Edit menus similar to those described above, but operate on individual resources or individual elements of a resource.

If you hold down the Option key when opening a resource, the **general data editor** is invoked—this allows you to edit the resource as hexadecimal data. If you hold down the Shift and the Option keys while opening, ResEdit shows you a list of all editors and templates.

Caution: Individual editors may not be appropriate for all resource types.

The menus for some of the editors are discussed below. The use of the remaining editors should be apparent when you run them.

Note: The general data editor will not edit resources larger than 16K bytes in length; however, you can *move* larger resources with the Cut, Copy, Paste, and Clear commands as described above.

CURS Resources

For 'CURS' resources, the editor displays three images of the cursor. All three images may be manipulated with the mouse.

The left image shows how the cursor will appear. The middle image is the mask for the cursor, which affects how the cursor appears on various backgrounds. The right image shows a gray picture of the cursor with a single point in black—this point is the cursor's hot spot.

The **Cursor** menu contains the following commands:

Try Cursor / Restore Arrow	Try Cursor lets you try out the cursor by having it become the cursor in use. Restore Arrow restores the standard arrow cursor.
Data –>Mask	Copies the cursor image to the mask editing area.

DITL Resources

For 'DITL' resources, the editor displays an image of the item list as your program would display it in a dialog or alert box. When you select an item, a size box appears in the bottom right corner of its enclosing rectangle so that you can change the size of the rectangle. You can move an item by dragging it with the mouse.

If you open an item within the dialog box, the editor associated with the item is invoked; for an 'ICON', for example, the icon editor is invoked. If you hold down the Shift and Option keys while opening, the 'DITM' editor is invoked instead—this is a special-purpose editor for editing items in an item list. If you hold down just the Option key while opening, the general data editor is invoked.

The **DITL menu** contains the following commands:

Bring to Front	Allows you to change the order of items in the item list. Bring to Front causes the selected item to become the last (highest numbered) item in the list. The actual number of the item is shown by the 'DITM' editor.
Send to Back	Like Bring to Front, except that it makes the selected item the first item in the list—that is, item #1.
Grid	Aligns the item on an invisible 8 pixel by 8 pixel grid. If you change the item location while Grid is on, the location will be adjusted such that the top left corner lies on the nearest grid point above and to the left of the location you gave it. If you change the size, it will be made a multiple of 8 pixels in both dimensions.

Use RSRC Rect	Restores the enclosing rectangle to the rectangle size stored in the underlying resource. Note that this works on 'ICON', 'PICT', and 'CNTL' items only; the other items have no underlying resources.
Resize Window	Adjusts the window size so that all items in the item list are visible in the window.

FONT Resources

For 'FONT' resources, the editor window is divided into three panes: a sample text pane, a character selection pane, and a character editing pane. These are shown in Figure 5.

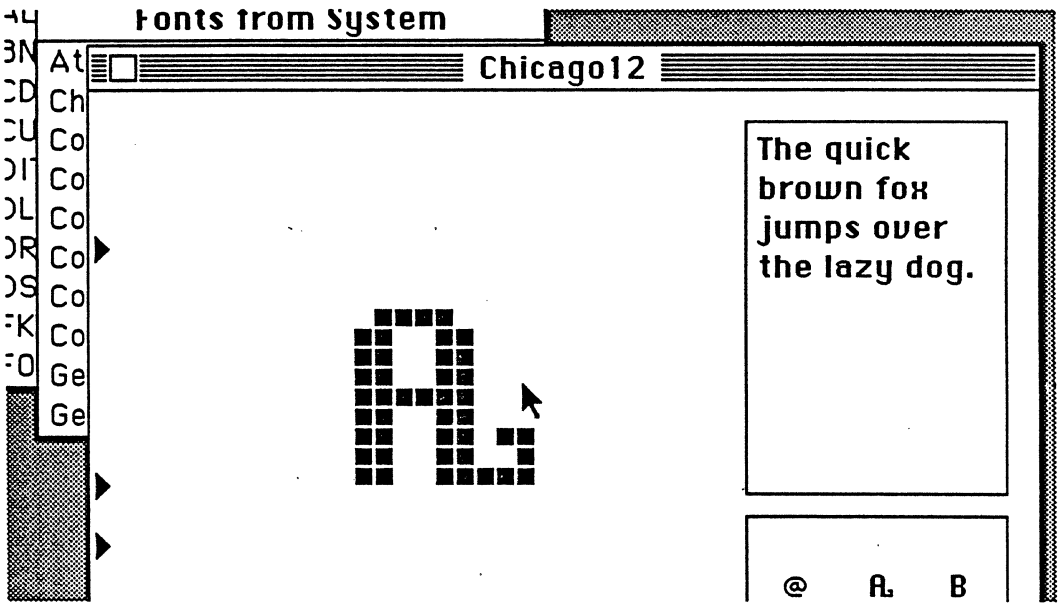

Figure 5. FONT Editor Window

The **sample text pane**, at the upper right, displays a sample of text in the font being edited. (You can change this text by clicking in the text pane and using normal Macintosh editing techniques.)

The **character selection pane** is below the text pane. You can select a character to edit by typing it (using the Shift and Option keys if necessary), or by clicking on it in the row of three characters shown. (Click on the right character in the row to move upward through the ASCII range; click on the left character to move downward.) The character you select is boxed in the center of the row with its ASCII value shown below it.

The **character editing pane** on the left side of the window shows an enlargement of the selected character. Like FatBits in MacPaint, it's edited by clicking bits on and off. The black triangles at the bottom of the character editing pane set the left and right bounds (i.e., the character width). The three triangles at the left of the pane control the ascent, baseline, and descent.

Caution: Changing the ascent or descent of a character changes the ascent or descent for the entire font.

Any changes you make in the character editing pane are reflected in the text pane and the character selection pane. Remember that you cannot save the changes until you quit.

You can also change the name of a font. The font name is stored as the name of the resource of that font family with size 0. This resource does not show up in the normal display of all fonts in a file. To display it, hold down the Option key when you open FONT from the file window. This will bring up the generic list of fonts. Select the font with the name you wish to change and choose Get Info.

ICN# Resources

For 'ICN#' resources, the editor displays two panes in the window. The upper pane is used to edit the icon. It contains an enlargement of the icon on the left and an enlargement of the icon's mask on the right. The lower pane shows, from left to right, how the icon will look unselected, selected, and open on both a white a gray background.

To install a new icon for your application when you already have an old one in the Finder's desktop file:

1. Open the file called DeskTop.
2. Open type 'BNDL' and find the bundle that is your application's. (This is the one that has your owner name in it.) Look through the bundle and mark down the type and resource ID of all resources bundled together by the bundle (i.e., the 'ICN#'s and 'FREF's).
3. Go back to the DeskTop window and remove these resources along with your 'BNDL' and signature resource (the resource whose type name = your creator type).
4. Now close the DeskTop window, save changes, and quit ResEdit. Your new icon will be installed.

Creating a Resource Template

You can customize ResEdit by creating new templates for your own resource types. The generic way of editing a resource is to fill in the fields of a dialog box—this is the way you currently edit 'FREF's, 'BNDL's, 'STR#'s, etc. The layout of these dialog boxes is set by a template in ResEdit's resource file. The template specifies the format of the resource and also specifies what labels should be put beside the editText items in the dialog box that's used for editing the resource. You can find these templates by opening the ResEdit file and then opening the type window for 'TMPL's. For example, if you open the template for 'WIND' resources (this is the 'TMPL' with name "WIND"), you'll see that they consist of the following, in the order listed:

- a RECT (4 words) specifying the boundary of the window
- a word that is the procID for the window (DWRD tells ResEdit to display the word in decimal as opposed to hex)

- a Boolean indicating whether or not the window is visible (BOOL is 2 bytes in the resource but is displayed as a radio button in the dialog window used for editing)
- another Boolean indicating whether or not the window has a close box
- a long that is the refCon for the window (DLNG indicates that it should be displayed in the editor as a decimal number)
- a Pascal string; the title of the window (PSTR)

You can look through the other templates and compare them with the structure of those resources to get a feeling for how you might define your own resource template.

The types you have to choose from for your editable data fields are:

DBYT, DWRD, DLNG — decimal byte, word, long

HBYT, HWRD, HLNG — hex byte, word, long

HEXD — hex dump of remaining bytes in resource

PSTR — a Pascal string (length byte followed by the characters)

LSTR — long string (length long followed by the characters)

WSTR — same as LSTR, but a word rather than a long

ESTR, OSTR — Pascal string padded to even or odd length (needed for DITLs)

CSTR — a C string

ECST — even-padded C string

OCST — odd-padded C string (padded with nulls)

BOOL — Boolean

BBIT — binary bit

TNAM — type name (like OSType and ResType, i.e., 4 characters)

CHAR — a single character

H*nnn* — 3-digit hex number (where *nnn* < $900); displays *nnn* bytes in hex format.

ResEdit will do the appropriate type checking for you when you put the editing dialog window away.

The template mechanism is flexible enough to describe a repeating sequence of items within a resource, as in 'STR#'s, 'DITL's, and 'MENU's. You can also have repeating sequences within repeating sequences as in 'BNDL's. To terminate a repeating sequence, put the appropriate code in the template as follows:

LSTZ-LSTE	terminated by a 0 byte (as in 'MENU's)
ZCNT/LSTC-LSTE	terminated by a zero-based count that starts the sequence (as in 'DITL's)
OCNT/LSTC-LSTE	terminated by a one-based count that starts the sequence (as in 'STR#'s)
LSTB-LSTE	ends at the end of the resource (no example exists in the given templates)

To create your own template:

1. Open the ResEdit file window.
2. Open the 'TMPL' type window.
3. Choose New from the File menu.
4. Select the list separator.
5. Choose New from the File menu. You may now begin entering the label,type pairs that define the template. Before closing the template editing window, choose Get Info from the File menu and set the name of the template to the name of your resource type.
6. Close the ResEdit file window and save changes.

The next time you try to edit or create a resource of this new type, you'll get the dialog box in the format you have specified.

How to Write an Add-on Editor

This section describes how to add an editor of your own type to ResEdit. It includes an outline for a sample editor, a description of the ResEd interface file, and explanations of the ResEd Resource Editor routines that you can use in your code.

There are two types of drivers: pickers and editors. The **picker** is the code that displays all the resources of one type in the resource type window. Pickers are given a resource type and should display all resources of that type in the current resource file, using a suitable display format. If the picker is given an open call *and* there's a suitable editor, it should launch the editor. The **editor** is the code that displays and allows you to edit a particular resource. The editor is given a handle to the resource object and should open an edit window (or windows) for the user. Pickers and editors are separate from the main code of ResEdit.

Note that pickers and editors can be opened from anywhere: For instance, a dialog editor might open an icon picker so that the user can choose an appropriate icon. And the user could, while in the icon picker, open the icon editor to create a new icon if desired.

Outline for a Sample Editor

The basic structure your editor should have is outlined below.

> *Note*: Routines defined in the ResEd interface file are shown in bold. 'XXXX' represents your resource type.

The philosophy of the sample editor is that you will first call the "EditBirth" routine when a new instance of your editor is needed. This routine is passed two handles: a handle to the resource to be edited (the same handle that would be received by using a GetResource call), and a handle back to the picker that has launched this editor. The editor should then create a window and set up any data structures needed to operate. Because the editor will be loaded in and out of memory during any given session and because the editor doesn't have access to global variables, you should create a handle to a data structure to hold all data that needs to be preserved between calls and store its handle in the edit data structure (rXXXXrec in the example). Note that the handle to the edit data structure is stored in the window's refCon parameter. The main program uses this to identify which subprogram is to receive a given call. The main program will determine which editor should receive which events, so you need to do very little event decoding in your editor. Also, during an update event, the BeginUpdate and EndUpdate calls are done by the main program, so don't do them in your editor.

There are several points to consider:

- When writing your editor, always know which resource you are requesting and where it will come from. Many resource files may be open at any given time, and you should always make sure which resource file you are accessing by using UseResFile or similar operations whenever a resource is needed.

- Remember that your editor may be called with an empty handle in order to create an entirely new instance of the type you edit.

After creating your editor, compile and link. Then use ResEdit to copy the 'RSSC' resource (and any other resources you have created for use by your editor) from the newly created file into ResEdit itself. ResEdit will then be able to edit using the new editor. When copying the resource across, be certain to check for and avoid conflicting resource ID numbers—it's a simple matter to check the ResEdit file (using ResEdit itself) to see which ID numbers (and 'RSSC' types) are already taken.

> *Note:* In the following sample editor, routines defined in the ResEd interface file appear in **bold.**

```
UNIT ResXXXXEd;

{XXXX Editor}
INTERFACE
USES {$U-}
     {$U obj/MemTypes   }    MemTypes,
     {$U obj/QuickDraw  }    QuickDraw,
```

```
    {$U obj/OSIntf    }    OSIntf,
    {$U obj/ToolIntf  }    ToolIntf,
    {$U obj/PackIntf  }    PackIntf,
    {$U obj/ListIntf  }    ListIntf,
    {$U obj/ResEd     }    ResEd;

{$OV-}{Overflow check off }
{$R-} {Range checking off }
{$U-} {Lisa Libraries off }
{$X-} {Stack expansion off}
{$M+} {Do this for Mac    }

TYPE
    rXXXXPtr    = ^rXXXXRec;
    rXXXXHandle = ^rXXXXptr;
    rXXXXRec    =
          RECORD
             father:    ParentHandle;  {Back ptr to dad    } {normal}
             name:      Str64;
             windPtr:   WindowPtr;     {This view's window} {normal}
             rebuild:   BOOLEAN;       {Set TRUE if things have been }
                                       { changed} {normal}
             hXXXX:     Handle;        {The resource we are working on}
             menuXXXX:  MenuHandle;    {our menu}
          END; {rXXXXRec}

PROCEDURE EditBirth (Thing:Handle; Dad:ParentHandle);
PROCEDURE PickBirth (t:ResType; Dad:ParentHandle);
PROCEDURE DoEvent (VAR Evt:EventRecord; MyXXXX:rXXXXHandle);
PROCEDURE DoInfoUpdate (oldID,newID:INTEGER; MyXXXX:rXXXXHandle);
PROCEDURE DoMenu (Menu,Item:INTEGER; MyXXXX:rXXXXHandle);

IMPLEMENTATION

CONST
    forever = FALSE;

TYPE
    dummy = INTEGER;

VAR
    dummy: INTEGER;



PROCEDURE EditBirth {Thing:Handle; Dad:ParentHandle};
VAR
    MyXXXX:  rXXXXHandle;
    w:       WindowPtr;
    s:       Str255;
Begin {EditBirth}
{ Prepare window title and request creation of a new window }
s := 'Window';
SetETitle (Handle (thing), s);
ConcatStr (s,' from ');
w := WindSetup (100,100,s,dad^^.name);
{If we got a new window, then start up the editor}
IF ORD(w) <> 0
```

```
THEN
     BEGIN
     FixHand (SIZEOF(WindowRecord),Handle (thing));
     {Make sure we have enough room in the handle for a complete      }
     { window record. (later this will be assigned to MyXXXX^^.hXXXX) }

     {Get memory for and handle to our instance record}
     MyXXXX := rXXXXHandle (NewHandle(SIZEOF(rXXXXRec)));

     HLock( Handle (MyXXXX) );
     WITH MyXXXX^^ DO
           BEGIN
           {Put information about this incarnation of the editor and  }
           { the window it is serving into our record.      }
           { (always passed around in the handle MyXXXX).}
           windPtr := w;
           father  := dad;
           hXXXX   := thing;
           {Let the main program know who is to manage this window by }
           { giving it both our resource ID number and our instance   }
           {record handle.  }
           WITH WindowPeek (w)^ DO
              BEGIN
              windowKind := ResEdID;
              refCon  := ORD(MyXXXX);
              END; {WITH}
           END;{WITH}
     { Set up menus,the view, etc. for this window }
     HUnlock ( Handle (MyXXXX) );
     END; {IF ORD(w)<>0}
END;  {EditBirth}


PROCEDURE PickBirth {t:ResType; Dad:ParentHandle};
BEGIN {PickBirth}
END;  {PickBirth}


PROCEDURE DoEvent {VAR Evt:EventRecord; MyXXXX:rXXXXHandle};
VAR
     MousePoint: Point;
     act:        BOOLEAN;
BEGIN {DoEvent}
BubbleUp ( Handle (MyXXXX) );   {Move our item up im memory}
HLock ( Handle (MyXXXX));       {Lock it down}
WITH MyXXXX^^ DO
     BEGIN
     {Handle event passed to us by main program.  Just like a 'real' }
     { event loop, except there is no loop and we don't have to      }
     { handle as much because the main program will do all the stuff }
     { that doesn't apply to us. }
     mousePoint := evt.where;   {Point at which the event occured}
     SetPort (windPtr);         {Set the port to our window}
     GlobalToLocal (mousePoint); {Convert event location to local coords}
     CASE evt.what OF
           mouseDown:  BEGIN
```

```
                    END; {mouseDown}
          activateEvt:BEGIN
                    AbleMenu (fileMenu,filetop);
                    act := ODD(evt.modifiers);
                        IF act
                          THEN
                             BEGIN {Activate event}
                             END   {Activate event}
                          ELSE
                             BEGIN {Deactivate event}
                             END;  {Deactivate event}
                          END {activateEvt};
          updateEvt:  BEGIN
                    END; {updateEvt}
          keyDown:    BEGIN
                    END; {keyDown}
          END; {CASE evt.what}
     END; {WITH MyXXXX^^}
HUnlock ( Handle (MyXXXX));
END; {DoEvent}


PROCEDURE DoInfoUpdate {oldID,newID: INTEGER; MyXXXX: rXXXXHandle};
VAR
     s:  Str255;

BEGIN {DoInfoUpdate}
WITH MyXXXX^^ DO
     BEGIN {Since our ID has changed, we need to change our window title}
     s := 'Window';
     SetETitle ( Handle (hXXXX), s);
     ConcatStr (s,' from ');
     ConcatStr (s,father^^.name);
     SetWTitle (windPtr,s);
     {Now, let our father object know that our ID has been changed}
     CallInfoUpdate (oldID,newID,father^^.wind^.refCon,
          father^^.wind^.windowkind);
     END; {WITH MyXXXX^^}
END; {DoInfoUpdate}


PROCEDURE DoMenu {Menu,Item: INTEGER; MyXXXX: rXXXXHandle};
VAR
     saveRefNum: INTEGER;

     PROCEDURE DoClose;
     BEGIN
     WITH MyXXXX^^ DO
          BEGIN
          CloseWindow (windPtr); {Close the window}
          WindFree (windPtr);    {Mark the window record as being available}
          InitCursor;           {Make sure the cursor is the arrow cursor }
          {Delete any menus that we added and redraw menu bar.  }
          { Be sure to dispose of any handles you are done with.}
          END; {WITH MyXXXX^^}
     DisposHandle ( Handle(MyXXXX));
```

```
    END; {DoClose}

BEGIN {DoMenu}
BubbleUp ( Handle (MyXXXX));
HLock ( Handle (MyXXXX));
WITH MyXXXX^^ DO
    BEGIN
    SetPort (windPtr); {Set the port to our window}
    {Again, we handle the menu stuff just as we would in a 'real' }
    { application except that we only have to handle those items  }
    { that apply to ourselves. }
    CASE Menu OF
          fileMenu: CASE Item OF
             CloseItem: BEGIN
                   DoClose; {Close our window}
                   EXIT (DoMenu); {Return to main program}
                   END; {CloseItem}
             RevertItem: BEGIN
                   {The area under window will need to be updated}
                   InvalRect (windPtr^.portrect);
                   {We'll need to restore the cur resource file }
                   { reference number when we're done here.}
                   saveRefNum := CurrentRes;
                   {We're going to be using the resource file we }
                   { came from.}
             UseResFile (HomeResFile ( Handle(hXXXX)));
             {Read in the old copy from disk (see documentation  }
             { for revertResource). Clear it out unless this     }
             { was a newly created resource, in which case don't.}
             IF NOT RevertResource ( Handle(hXXXX))
             THEN
                BEGIN
                RmveResource (Handle(hXXXX));
                MyXXXX^^.father^^.rebuild := TRUE;
                DoClose;
                EXIT (DoMenu);
                END; {IF NOT RevertResource... }
             {Go back to using old resource file.}
             UseResFile(saveRefNum);
             END; {RevertItem}
          GetInfoItem:
             BEGIN
             ShowInfo (Handle(hXXXX),ParentHandle(MyXXXX));
             END; {GetInfoItem}
          END; {FileMenu: CASE Item OF}
    EditMenu: CASE Item OF
             CutItem  :;
             CopyItem :;
             PasteItem:;
             ClearItem:;
             END; {EditMenu: CASE Item OF}
    END; {CASE Menu OF }
END; {WITH MyXXXX^^}
END; {DoMenu}
END.
```

Resource Compiler Input File

```
* Resource Compiler input file for a resource editor

RSRC/ResXXXX.rsrc


type RSSC= DRVR
    obj/resXXXXed!@XXXX, nnn {Substitute an appropriate ID for nnn}
```

ResEd Interface File

The ResEd file should be USEd by any Pascal implementation of an add-on picker or editor. The companion file ResEd68k should be linked with the Pascal to realize the partially linked object file for inclusion in the ResEdit.RSRC file.

Resource Editor Routines from ResEd

The following two routines are used to launch resource type pickers and editors.

```
PROCEDURE CallPBirth (theType: ResType; parent: ParentHandle; id:
INTEGER);
```

Launches a picker for a given resource. An editor will have little use for this routine unless you want your user to be able to choose other resources to edit from within your editor. Notice that there is no communication back to the caller as to which item was chosen, as it is the job of a picker to launch the editor for the item chosen.

```
PROCEDURE CallEBirth (theResource: Handle; parent: ParentHandle; id:
   INTEGER);
```

Launches an editor for a specific resource. *theResource* is the handle to that resource as returned by GetResource and *parent* is the handle to your own picker record. *id* is the resource ID of the resource being edited. This can be a handy routine to call if your editor also needs to edit other types (for instance, to edit an ICON from within a DITL). If you use this routine, your editor must also act like a picker—this means that the first portion of your record must be in the format of a picker record (defined in the ResEd interface file), and that you must respond to the changed status passed back from the editor.

The following routines are used to feed events and menu calls to the appropriate code segments.

```
PROCEDURE CallInfoUpdate (oldID,newID: INTEGER; object: LONGINT; ID:
   INTEGER );
```

Tells the picker that launched your editor that something has changed that makes it necessary to rebuild the picker list (that is, an ID or name has been changed, added, or deleted).

```
PROCEDURE PassMenu ( menu, item: INTEGER; father: ParentHandle );
```

Passes menu selections on to any son pickers or editors that you have launched with CallEBirth or CallPBirth. Here is an example of its use:

```
PassMenu(file,close,myObj);    { Tell all subsidiary processes }
                               { to close}
```

Note: Most editors won't use the following event- and menu-handling routines.

```
PROCEDURE CallEvent ( VAR evt: EventRecord; object: LONGINT; id:
   INTEGER );
```

Passes events on to pickers or editors that you have launched with CallEBirth or CallPBirth. *object* identifies the resource and editor; *id* identifies the resource ID of the object.

```
PROCEDURE CallMenu ( menu, item: INTEGER; object: LONGINT; ID:
   INTEGER);
```

Passes menu selections on to pickers or editors that you have launched with CallEBirth or CallPBirth.

Common Utilities Exported by ResEdit

These routines are used for handling windows.

```
FUNCTION WindAlloc: WindowPtr;
```

Returns a pointer to a window record (actually a dialog record) to be used by your editor. Using this routine instead of allocating your own window pointer can help to reduce heap fragmentation.

```
PROCEDURE WindFree ( w: WindowPtr );
```

Releases the window pointer allocated by WindAlloc. Use this when your editor is quitting and you are done with the window.

```
FUNCTION  WindList (w: WindowPtr; nAcross: INTEGER; pt:Point;
   drawProc:INTEGER): ListHandle;
```

Creates a new empty list and returns a handle to that list. (For more information on lists, see the "List Manager" chapter in Volume IV of *Inside Macintosh.*)

```
PROCEDURE WindOrigin ( w: WindowPtr );
```

Places the window pointed to by *w* at an offset down and to the right of the current front window.

```
FUNCTION WindSetup (width,height: INTEGER; t,s: STR255): WindowPtr;
```

Creates and automatically positions a new window with the given width and height, and with the title of ConcatStr(*t*,*s*);

Extended Resource Manager Routines which Act on a Single Resource File

The following five routines are the same as their corresponding routines described in *Inside Macintosh*, except that they operate only on the currently selected resource file and they always load the resource into the application heap. (They are selected by UseResFile, etc.)

Note: These routines are provided in the 128K ROM; they are provided here for compatibility with 64K ROMs.

```
FUNCTION  Count1Type: INTEGER;

PROCEDURE Get1IndType (VAR theType: ResType; index: INTEGER);

FUNCTION  Count1Res (theType: ResType ): INTEGER;

FUNCTION  Get1IndResource (theType: ResType; index: INTEGER ):
   Handle;

FUNCTION  Get1Resource (theType: ResType; id: INTEGER ): Handle;


FUNCTION  GetResLoad: BOOLEAN;
```

Returns the current state of SetResLoad.

```
PROCEDURE Get1MapEntry (VAR theEntry: ResMapEntry; t: ResType; id:
   INTEGER);
```

Note: Map entry is the same as "resource reference" in I.M.

```
PROCEDURE Get1IMapEntry (VAR theEntry: ResMapEntry; t:ResType; index:
INTEGER);

PROCEDURE GiveEBirth (h: Handle; pick: PickHandle);


FUNCTION RevertResource ( h: Handle ): BOOLEAN;
```

Given a handle to a resource that you are editing, this routine restores it to the state it was in before the editing started.

Miscellaneous Routines

```
PROCEDURE AbleMenu ( menu: INTEGER; enable: LONGINT );
```

Given a menu in *menu* and a mask in *enable* (see the interface file) this routine enables and disables menu items. (AbleMenu differs from the Resource Manager routines EnableItem and DisableItem in that it acts on the entire menu.)

```
PROCEDURE AppRes;
```

This routine does a UseResFile on the Resource editor itself. This routine is useful if you need to get a resource from the resource editor, such as an ICON or DITL, for your editor to use.

```
FUNCTION AddNewRes (hNew: Handle; t: ResType; idNew: INTEGER; s:
   Str255) :BOOLEAN;
```

Adds a new resource. AddNewRes returns TRUE if successful, and reports an error to the user if it fails. This is the same as calling AddResource + ResError.

```
PROCEDURE BubbleUp ( h: Handle );
```

Performs the Memory Manager routine MoveHHi with additional error reporting.

```
FUNCTION BuildType ( t: ResType; l: ListHandle ): INTEGER;
```

Given a list that has been initialized with no rows (see the WindList routine above), this routine builds a list of all resources of type *t* from the current resource file. If SetResLoad(TRUE) has been called, all the resources will be loaded in also. It returns a count of the number of instances now in the list.

```
PROCEDURE ClearHand ( h: Handle );
```

Clears the data indicated by the handle *h* to all zeros.

```
FUNCTION CopyRes ( VAR h: Handle; makeID: BOOLEAN; refNum: INTEGER)
   :Handle;
```

Given a handle to a resource, CopyRes makes a copy of the resource to the resource file specified by *refNum*. Note that the handle *h* is changed, so don't keep track of your resource by saving its handle before using this call. If *makeID* is true, then a unique ID will be assigned to the copy; otherwise it retains the ID of the original. CopyRes returns a handle to the new copy (in the new file).

```
PROCEDURE ConcatStr ( VAR str1: Str255; str2: Str255 );
```

Concatenates *str2* to *str1*.

```
PROCEDURE DoListEvt (theEvent: EventRecord; l: ListHandle);
```

Given an event, this routine does the standard dispatch to the List Manager (see Volume IV of *Inside Macintosh*). The port must be set to the window that owns the event. DoListEvt also enables the File menu and draws controls in the window.

```
FUNCTION DupPick (h: Handle; c: cell; pick: PickHandle): Handle;
```

Takes a resource and duplicates it; adds it to the picker list handle passed; and does an InvalRect on thePort for the new cell. It also makes the new cell the selection.

```
FUNCTION ErrorCheck (err,msgID: INTEGER): BOOLEAN;
```

Given a result code and a message ID, this routine brings up an error dialog if the result code is nonzero. If *msgID* is negative, it is a fatal error message and is retrieved from 'STR#' resource ID = 128; otherwise it is retrieved from 'STR#' resource ID = 129. Be sure to add your new strings to the end of the existing list of the 'STR#'.

```
FUNCTION FileNewType (types: ListHandle; VAR s: Str255): BOOLEAN;
```

Puts up a dialog with a list of the types of resources that can be edited. The routine returns TRUE if a type is being returned, FALSE if Cancel was clicked. The type is returned in *s*.

```
PROCEDURE FixHand ( s: LONGINT; h: Handle );
```

Makes sure the object that *h* to which is a handle is *s* bytes long. If it's longer, FixHand shrinks it; if it's shorter, FixHand expands it and fills the new part with zeros.

```
FUNCTION HandleCheck ( h: Handle; msgID: INTEGER ): BOOLEAN;
```

Checks to see if the handle is nil or empty. If it is either, HandleCheck returns error *msgID* to the user. It returns TRUE if the handle is OK, and FALSE if there's an error.

```
PROCEDURE MetaKeys ( VAR cmd, shift, opt: BOOLEAN );
```

Returns the modifier flags for the last event.

```
FUNCTION NewRes ( s: LONGINT; t: ResType; l: ListHandle; VAR n:
   INTEGER) :Handle;
```

Given a size *s*, this routine allocates a new handle, clears it, adds it to the current resource file as a resource of type *t*, adds it to the list *l*, and returns the handle to the new resource.

```
PROCEDURE PickEvent ( VAR evt: EventRecord; pick: PickHandle );
```

Handles all events for standard pickers. Call it from your picker's event procedure.

```
PROCEDURE PickInfoUp ( oldID,newID: INTEGER; pick: PickHandle );
```

Handles all info updates for standard pickers. Call it from your picker's doInfoUpdate procedure.

```
PROCEDURE PickMenu ( menu, item: INTEGER; pick: PickHandle );
```

Handles all menu selections for standard pickers. Call it from your picker's DoMenu procedure.

```
FUNCTION  ResEdID: INTEGER;
```

Returns the resource ID of the editor/picker that is calling this routine.

```
PROCEDURE ResEverest;
```

Sets the current resource file to the last one opened.

```
PROCEDURE ScrapCopy ( VAR h: Handle );
```

Copies the resource identified by *h* from the ResEdit scrap.

```
PROCEDURE ScrapEmpty;
```

Empties the ResEdit scrap. Call this before doing a paste.

```
PROCEDURE ScrapPaste ( resFile: INTEGER );
```

Copies the resource identified by *h* to the ResEdit scrap.

```
PROCEDURE SetResChanged (h: Handle);
```

Marks the resource *h* as changed so that it will be updated when quitting.

```
PROCEDURE SetETitle ( h: Handle; VAR str: STR255 );
```

Given a handle to a resource, this routine places its ID and name into *str* (concatenating them).

```
FUNCTION  ResEditRes: INTEGER;
```

Returns the resource ID of the resource editor.

```
PROCEDURE ShowInfo (h: Handle; dad: ParentHandle);
```

Puts up a GetInfo window for the given resource that belongs to the given father object.

```
PROCEDURE SeedFill (srcPtr,dstPtr: Ptr; srcRow,dstRow,height,words:
   INTEGER; seedH,seedV: INTEGER);
```

Given a source and a destination bit image along with a seed location, this routine fills the bits in the destination with black.

```
PROCEDURE CalcMask (srcPtr,dstPtr: Ptr; srcRow,dstRow,height,words:
   INTEGER);
```

Calculates a mask for the given source bit image and puts it into the destination.

Constants

```
CONST
{Standard menus exported by the resource editor shell}
      FileMenu    = 2;
         NewItem     = 1;
         OpenItem    = 2;
         OpnOther    = 3;
         OpnGnrl     = 4;
         CloseItem   = 5;
         RevertItem  = 6;
         GetInfoItem = 7;
         QuitItem    = 9;

      EditMenu    = 3;
         UndoItem    = 1;
         CutItem     = 3;
         CopyItem    = 4;
         PasteItem   = 5;
         ClearItem   = 6;
         DupItem     = 8;

      fileAll       = $FFFFFEBF;   { All enabled }
      fileNoOpen    = $FFFFFEE3;   { open disabled - for when editor on top}
      fileTop       = $FFFFFEE1;   { Close, GetInfo, Revert, Quit enabled }
      fileClose     = $FFFFFE21;   { Close, Quit enabled }
      fileNoRevert  = $FFFFFEBF;   { No revert }
      fileNoInfo    = $FFFFFE7F;   { No get info }
      fileOpQuOnly  = $FFFFFE1D;   { Open and Quit enable }
      fileQuit      = $FFFFFE01;   { Only Quit enabled }
      fileNoAsMask  = $FFFFFFF7;   { Mask off the Open as... selection }

      editAll       = $FFFFFF7B;   { All enabled }
      editNoUndo    = $FFFFFF79;   { All enabled except undo }
      editNoDup     = $FFFFFE79;   { All enabled except undo and dup. }
      editNone      = $FFFFFE01;   { None enabled }
      editAcc       = $0000007B;   { Common enabled for desk acc. }
      editCopy      = $FFFFFE11;   { Only copy enabled }
```

Data Types

```
TYPE
      STR64 = STRING[64];

      {map entry def for new resource manager call}
      ResMapEntry =
         RECORD
            rID:       INTEGER;
            rNameOff:  INTEGER;
            rLocn:     LongInt;
            rHndl:     Handle;
         END; {ResMapEntry}
```

Each driver has its own object handle. This handle has to start with a handle to its parent's object, followed by the name distinguishing the father. This name will be part of the son's window title. The next field should be the window of the object, which may be used by son to get back to the father (through the refCon in the windowRec). The rest of the handle can be of any format.

```
      ParentPtr  = ^ParentRec;
      ParentHandle  = ^ParentPtr;
      ParentRec  =
         RECORD
            father:  ParentHandle;
            name:    Str64;
            wind:    WindowPeek;
            rebuild: BOOLEAN;   {flag set by son to indicate that world}
                                { has changed so father should rebuild }
                                { list }
         END;
```

The standard picker record is shown below:

```
      PickPtr     = ^PickRec;   {Any type is OK here  }
      PickHandle  = ^PickPtr;
      PickRec =
         RECORD
            father:     ParentHandle;  {Back ptr to dad}
            fName:      Str64;
            wind:       WindowPtr;     {Picker window}
            rebuild:    BOOLEAN;
            pickID:     INTEGER;       {Resource ID of this picker}
            rType:      ResType;       {Type for picker}
            rNum:       INTEGER;       {resfile number}
            rSize:      LONGINT;       {size of a null resource}
            nInsts:     INTEGER;       {Number of instances}
            instances:  ListHandle;    {List of instances}
            drawProc:   Ptr;           {List draw proc.}
            scroll:     ControlHandle; {Scroll bar}
         END;
```

Summary of ResEd Routines

Routines used to give birth to resource pickers and editors:

```
PROCEDURE CallPBirth (t: ResType; parent: ParentHandle; id:
   INTEGER);
PROCEDURE CallEBirth (thing: Handle; parent: ParentHandle; id:
   INTEGER);
```

Routines used to feed events and menu calls to the appropriate code segments:

```
PROCEDURE CallEvent (VAR evt: EventRecord; object: LONGINT; id:
   INTEGER);
PROCEDURE CallMenu (menu, item: INTEGER; object: LONGINT; id:
   INTEGER);
PROCEDURE CallInfoUpdate (oldID,newID: INTEGER; object: LONGINT;
   id: INTEGER);
PROCEDURE PassMenu ( menu, item: INTEGER; father: ParentHandle );
```

Common utilities exported by ResEdit:

```
FUNCTION  WindAlloc: WindowPtr;
PROCEDURE WindFree ( w: WindowPtr );
FUNCTION  WindList ( w: WindowPtr; nAcross: INTEGER; pt:Point;
   drawProc: INTEGER): ListHandle;
PROCEDURE WindOrigin ( w: WindowPtr );
FUNCTION  WindSetup ( width, height: INTEGER; t, s: STR255 ):
   WindowPtr;
```

Extended Resource Manager routines which act only on one resource file:

```
FUNCTION  CurrentRes: INTEGER;
FUNCTION  Count1Res ( t: ResType ): INTEGER;
FUNCTION  Count1Type: INTEGER;
FUNCTION  Get1Index ( t: ResType; index: INTEGER ): Handle;
FUNCTION  Get1Resource ( t: ResType; id: INTEGER ): Handle;
PROCEDURE Get1IndType ( VAR theType:  ResType; i:  INTEGER);
FUNCTION  GetResLoad: BOOLEAN;
PROCEDURE Get1MapEntry (VAR theEntry: ResMapEntry; t: ResType; id:
   INTEGER);
PROCEDURE Get1IMapEntry (VAR theEntry: ResMapEntry; t: ResType;
   index: INTEGER);

PROCEDURE GiveEBirth (h: Handle; pick: PickHandle);
FUNCTION  RevertResource ( h: Handle ): BOOLEAN;
```

Miscellany:

```
PROCEDURE AbleMenu ( menu: INTEGER; enable: LONGINT );
PROCEDURE AppRes;
FUNCTION  AddNewRes (hNew: Handle; t: ResType; idNew: INTEGER; s:
   str255): BOOLEAN;
PROCEDURE BubbleUp ( h: Handle );

FUNCTION  BuildType ( t: ResType; l: ListHandle ): INTEGER;
PROCEDURE ClearHand ( h: Handle );
FUNCTION  CopyRes ( VAR h: Handle; makeID: BOOLEAN; resNew:
   INTEGER): Handle;
PROCEDURE ConcatStr ( VAR str1: STR255; str2: STR255 );
```

```
PROCEDURE DoListEvt ( e: EventRecord; l: ListHandle );
FUNCTION  DupPick ( h: Handle; c: cell; pick: PickHandle ): Handle;
FUNCTION  ErrorCheck ( err, msgID: INTEGER ): BOOLEAN;
FUNCTION  FileNewType (types: ListHandle; VAR s: str255): BOOLEAN;

PROCEDURE FixHand ( s: LONGINT; h: Handle );
PROCEDURE GetStr ( num, list: INTEGER; VAR str: STR255 );
FUNCTION  GetThePort: GrafPtr;
FUNCTION  HandleCheck ( h: Handle; msgID: INTEGER ): BOOLEAN;

PROCEDURE MetaKeys ( VAR cmd, shift, opt: BOOLEAN );
FUNCTION  NewRes ( s: LONGINT; t: ResType; l: ListHandle; VAR n:
   INTEGER): Handle;
PROCEDURE PickEvent ( VAR evt: EventRecord; pick: PickHandle );
PROCEDURE PickInfoUp ( oldID,newID: INTEGER; pick: PickHandle );

PROCEDURE PickMenu ( menu, item: INTEGER; pick: PickHandle );
FUNCTION  ResEdID: INTEGER;
PROCEDURE ResEverest;
PROCEDURE ScrapCopy ( VAR h: Handle );

PROCEDURE ScrapEmpty;
PROCEDURE ScrapPaste ( resFile: INTEGER );
PROCEDURE SetResChanged (h: Handle);
PROCEDURE SetETitle ( h: Handle; VAR str: STR255 );

FUNCTION  ResEditRes: INTEGER;
PROCEDURE ShowInfo (h: Handle; dad: ParentHandle);
PROCEDURE SeedFill (srcPtr,dstPtr: Ptr; srcRow,dstRow,height,words:
   INTEGER; seedH,seedV: INTEGER);
PROCEDURE CalcMask (srcPtr,dstPtr: Ptr; srcRow,dstRow,height,words:
   INTEGER);
```