

AUTODIALOG

by

John R. McMullen

JAM Software Pty Ltd

© Copyright 1986

ABOUT THIS MANUAL

AutoDialog is a "programmer-friendly" interface to the dialog manager. This manual describes the AutoDialog routines in detail.

You should already be familiar with

- Resources
- the toolbox Event Manager
- the Control Manager
- TextEdit
- the Dialog Manager.
- the List Manager

ABOUT AUTODIALOG

AutoDialog is a set of routines and data types that provide an easy way to run dialogs in a standard way, able largely to be specified by the resource editor instead of by programmer code. Its benefits are:

1. Cuts down code size if several complicated dialogs are used in one program. Cuts down programming time enormously. Calls such as `getCtlValue`, `setCtlValue`, `getDItem`, and `HiliteControl`, used to read the state of checkboxes and to dim and undim buttons, are eliminated from your code, being centralized in the `autoDialog` unit.
2. Handles, automatically and in a standard way, common item types lumped together as "user items" by the Macintosh Dialog Manager. These new types include
 - (a) gray lines drawn in the dialog;
 - (b) outlining of the default button;
 - (c) scrollable lists (handled through the List Manager package);
 - (d) "radio icons": groups of icons that behave like radio buttons.
3. Provides a standard `filterProc` to handle all the above new items, (particularly lists) as well as standard radio button

groups. Thus the programmer's filterproc need concern itself only with non-standard matters.

4. Allows the types and initial state of the dialog's items to be read in as a resource, facilitating development through the Resource Editor not only of the dialog's appearance, but a great deal of its behavior as well. Many dialogs can be run without writing a special-purpose filter routine.

5. Provides for automatic centering of dialogs on the screen of the machine it is running on.

6. Unlike the standard filtering procedure of the dialog manager, checks for command keys and invokes the appropriate editing responses.

7. Any item (not necessarily item #1) can be specified as the "Bold Item". The standard filtering procedure makes return/enter correspond to this bold button, if any.

WORKING WITH AUTODIALOG

The main aid to your understanding of AutoDialog is the source code of the program **AutoDemo**, supplied on the AutoDialog disk. To use AutoDialog you must:

1. From Lisa Pascal, include the line:

```
'USES {$U Autodialog} AutoDialog'
```

in your source code, and link explicitly with the file Autodialog[.OBJ]. From TML Pascal, you should include the two lines:

```
{ $I adTypes.ipas }  
{ $I adIntf.ipas }
```

in your source file, which will make the AutoDialog constants, data types and routines available in your program, and automatically include the correct library file name in your linker input file for you (you may have to edit the file names in these files to accord with the names of your own disks and folders). From assembly language, just link with the Lisa file AutoDialog.OBJ, or the Macintosh file AutoDialog.Rel, as the case may be.

2. Create, for each dialog that you want to run with autoDialog, a DSta resource with the same ID as the DLOG resource (see below).

3. Instead of calling getNewDialog, call adInit. Additional initialization can be done through calls to autoDialog's utility routines DimlItems, setDrawProc. In rare cases when the standard autoDialog filtering is not quite appropriate, you can also call setClickHook to specify extra handling for each special item in your dialog, and setKeyHook for special treatment of typed keys.

4. For modal dialogs, instead of passing NIL or a pointer to your own procedure to modalDialog, pass @adFilter. For modeless dialogs include in your main event loop code such as:

```
If GetNextEvent(EveryEvent,Event) then
  If IsDialogEvent(Event) then with event do begin
    if (what = ActivateEvt) or (what = updateEvt) then
      aDialog := dialogPtr(message)
    else
      aDialog := dialogPtr(frontWindow);
    itemHit := -1; {we need this for modeless dialogs}
    if (not adFilter(aDialog, Event, itemHit)) then
      if dialogSelect(Event, aDialog, itemHit) then;
    end
  else begin
    {your own processing}
  end;
```

5. Instead of calling disposDialog, call adDispose, which takes care of autoDialog's special structures as well.

CREATING 'DSta' RESOURCES WITH ResEdit

For every dialog you want to run with autoDialog, you must create a resource of type 'DSta', having the same resource ID as your 'DLOG' resource. This resource can be created with the application ResEdit, supplied by Apple to its developers.

A template resource (type 'TMPL') has been provided on the supplied disk, in a plain-iconed document file named 'AD template'. This has been provided to allow creation of the resource of type 'DSta' that autoDialog needs to initialize its dialogs. To install this template into your ResEdit application, carry out the following steps:

1. Open your ResEdit application, by clicking twice on its icon.
2. Open the file called 'AD Template'.
3. Double click on the 'TMPL' resource type.
4. Select 'Copy' from the Edit menu.
5. Close the 'AD Template' window.
6. Open the file 'ResEdit'.
7. Scroll until you see the 'TMPL' resource type.
8. Double click on the 'TMPL' resource type.
9. Select 'Paste' from the Edit menu.
10. Close the 'ResEdit' window and click 'Yes' when asked if you want to save.

You have just taught ResEdit how to create and edit 'DSta' resources.

To create a new 'DSta' resource for your application, carry out the following steps:

1. Create your dialog and dialog item list in the usual way, and decide what resource ID you want it to have.
2. Using ResEdit, and with the window of your resource file as the active window, select 'New' from the file menu.
3. Scroll the list of resource types until you see 'DSta' (Not there? Try the previous sequence of steps again).
4. Click on 'DSta' and click 'OK'. A new empty window titled 'DStas from myFile' will open.
5. Again, select 'New' from the File menu. You will be able to edit your DSta resource in a fairly self-explanatory way.
6. Close the resource window, and with your new resource selected in the 'DStas from...' window, select 'Get Info' from the File menu. Change its ID to be the same as that of your DLOG.

AUTODIALOG DATA TYPES

CONST

```
MaxDItem = 63;           {Largest allowed item number }
MaxBitBytes = 7;         {One less than the number of bytes needed
                           for an adSet structure = (MaxDItem + 1)
                           div 8 - 1
                           currently needs 8 bytes of bits}
MaxRGroup = 3;           {One less than the number of radio button
                           groups allowed}
MaxLItem = 1;            {One less than the number of scrollable
                           lists allowed}
```

TYPE

```
adItem = integer;
adSet = packed array [0..MaxBitBytes] of byte;
        {test membership with the Toolbox Utility 'BitTst' call}
procArray = array[0..MaxLItem] of procPtr;
        {for passing to adInit}
adListRecord = packed record      {8 bytes}
    adLInum: byte; {If non-zero, item number of list}
    adLButNum: byte; {if non-zero, item number of button equiv to
                      double click}

    adFiller: byte;
    adSelFlags: byte; {selection flags for list manager}
    adLIType: ResType; {parameter passed to list's dataFetch routine}
end;

adState = record
    adBoldItem: adItem;
        {If non-zero, the button to be outlined, and to which
         return/enter should correspond}
```

```

adList: array[0..maxLItem] of adListrecord;
      {Info about each scrollable list item. 8 bytes per item}
adRadioSet: array[0..MaxRGroup] of adSet;
      {Specifies which item numbers belong to the radio button
      groups}
adRIconSet: array[0..MaxRGroup] of adSet;
      {Specifies which item numbers belong to the radio icon
      groups}
adRadioOn: array[0..MaxRGroup] of adItem;
      {The item number of the radio button in each group which
      is currently on}
adRIconOn: array[0..MaxRGroup] of adItem;
      {The item number of the radio icon in each group which
      is currently on}
adCheckOn: adSet;
      {The item numbers of check boxes which are
      currently checked}
adGrayLines: adSet;
      {The item numbers which are "gray line" user items}
adDimSet: adSet;
      {The set of controls (radio, check, buttons) which are
      currently dim, ie 255 hiliting}
adItemCount: integer;
adEditID: integer;
      {if non-zero, menu ID of edit menu}
adRefCon: longInt;
      {use this for whatever you want}
adKeyHook: procPtr;
      {If non-NIL, pointer to keyHook routine}
adClickHook: procPtr;
      {If non-NIL, pointer to clickHook routine}
adHandle: array[1..MaxDItem] of handle;
      {handles to all the items - actually dynamic: only as much
      storage as is needed is used}

end;
adStatePtr = ^adState;
adStateHandle = ^adStatePtr;

```

DESCRIPTIONS OF AUTODIALOG ROUTINES

PROCEDURE adInit

(VAR theDialog: dialogPtr; theID: integer; centered: BOOLEAN; editID: integer;
fetchProcs: procArray);

Call adInit once (passing a NIL pointer for theDialog) instead of getNewDialog in order to read in your dialog from the resource file and initialize it. This procedure must be called before any other autoDialog routines, except for the general utilities centerDialog, centerAlert, setDrawProc, and getResNames.

First, by calling getNewDialog, adInit puts up a dialog as the front window, centering it on the machine's screen if centered is TRUE, and allocates its storage on the heap. Then adInit reads in the initial state of the dialog (it does this by doing a getResource('DSta', theID). The information from this resource, if any, is written into the appropriate fields of the daState record in

a new relocatable block; if no such resource is found, the new block is initialized with all fields except `adItem`Count set to zero). The handle to this block is stored in the `refCon` field of the dialog window record.

If theDialog is not NIL on entry, `adInit` merely resets the dialog to its initial state, as prescribed by the 'DSta' resource.

If you pass 0 for `editID`, `AutoDialog` will ignore the command key when the user types a key. If, however, you pass the menuID of your edit menu, `AutoDialog` will correctly interpret editing key commands for you, (doing calls to `TEFromScrap`, `TEToScrap`, and `ZeroScrap` as appropriate).

The array `fetchProcs` should contain a procedure pointer for each list item in the dialog. The routines in the array should be routines which provide strings to place in the lists. Note that a zero value in the `adList` field of the 'DSta' resource overrides a non-NIL pointer passed in this parameter. Each of these procedures should have the declaration:

```
PROCEDURE dataFetch(theParam: resType; VAR theIndex: integer; VAR theString:
str255);
```

This procedure will be called repeatedly by the `adInit` routine, until `dataFetch` returns a null string. The strings will be placed in the list, in sorted order (using `IUMagString`). TheParam will be read from the `adLType` array of the DSta resource. The `theIndex` parameter will be 1 on the first call, and will be incremented between successive calls. Your `dataFetch` routine can ignore this parameter, or manipulate as it as required.

The `AutoDialog` unit provides one standard routine of this form: `PROCEDURE getResNames` returns successively the names of resources of the type indicated in theParam in the current resource file, by calling `getIndResource(theParam, theIndex)`, incrementing `theIndex` as much as necessary to avoid returning resources not in the current resource file.

Finally, `adInit` sets the cursor to an arrow.

If no dialog state resource is found with the given resource ID, or not enough memory exists for the `setHandleSize` call, then `adInit` does nothing and returns with theDialog set to NIL. Otherwise, theDialog contains the dialog pointer of the initialized dialog.

```
PROCEDURE adDispose(theDialog: dialogPtr);
```

This disposes of all list items and releases the dialog state handle, before calling `disposDialog`. Call `adDispose` when you are completely through with a dialog that you initialized with `adInit`.

```
FUNCTION adFilter(theDialog: DialogPtr; VAR theEvent: EventRecord; VAR itemHit:
integer): BOOLEAN;
```

The `adFilter` function is a general-purpose filter routine, which can be passed as a parameter in your call to `modalDialog` (see "Handling Dialog Events" in the Dialog manager chapter of Inside Macintosh). Most dialogs will not require any further filtering.

For dialogs with items requiring extra processing, you can call `setClickHook` to specify an additional procedure for each such item.

Sometimes the state of the dialog must change according to whether a given `editText` item contains any text. For example, you may want to undim the "Save" button if and only if a non-null filename has been typed. If that is so, you can write a routine to test the `textEdit` item and do the appropriate dimming, and then pass its pointer to the `setKeyHook` procedure.

WARNING: If you dim or undim items in your `keyHook` or `clickHook` procedures, you must do so by calling the utility routine `dimItems`. Otherwise the fields in the state record may not be set correctly.

When called by the dialog manager, `adFilter` handles events as follows.

- In response to a keypress, if the `adBoldItem` of the dialog state record is non-zero, `adFilter` checks for return and enter, and if one of these keys was pressed, calls the `ClickHook` for `adBoldItem` (if one was set), returns `TRUE` (unless `keyHook` is set), and reports a click in the `adBoldItem`.

If the command key was held down, `adFilter` calls `dlgPaste`, `dlgCut`, `dlgCopy`, or `dlgDelete` as appropriate, and returns `TRUE` (unless `keyHook` is set) if one of these routines was called.

Otherwise, `adFilter` returns `FALSE` (unless `keyHook` is set).

If a `KeyHook` was set (by a previous call to `setKeyHook`), this is called before returning to the Dialog Manager. The value returned overrides the value that would otherwise be returned for `adFilter`. (However, the `KeyHook` routine is NOT called if return or enter is pressed when a default "bold" item has been specified. The clickhook for the bod item is called instead. Nor is it called if edit command keys are involved).

- In response to an activate event, `adFilter` returns `FALSE`. First, however, `adFilter` calls `LActivate` for each list item.
- In response to an update event, `adFilter` returns `FALSE`. First, however, `adFilter` draws all the radio icons that are "on", and calls `validRect` for each such radio icon. This is necessary, as the dialog manager will otherwise draw the icons in their normal "unselected" state. Then `adFilter` updates the lists and draws their borders.
- If the mouse button is pressed in an active control, `adFilter` calls the Control Manager function `TrackControl`. If the mouse button is released outside the control, `adFilter` changes the event to a `nullEvent` and sets `theItem` to -1, and returns `FALSE`. Otherwise, if there is no `clickHook` procedure for the control, `adFilter` returns `TRUE`. If there is a `clickHook` procedure, it is called before returning.

If the mouse button is pressed in a radio icon, `adFilter` simulates the Control Manager's routine `TrackControl`. If the mouse button is released inside the icon, `adFilter` changes the icon "selected" in the

group of icons, but returns FALSE (However, if there is a clickHook procedure , it is called before returning).

- If the mouse button is pressed in a list, adFilter calls the List Manager routine LClick, which handles scrolling and selecting in the list. If the List Manager reports a double click in a cell of the list, and if the list's field adLButNum in the dialog state record is non-zero, adFilter sets itemHit to adLButNum, calling its clickHook, if appropriate. Otherwise, if there is no clickHook procedure for the list (or adLButNum), adFilter returns TRUE. If there is a clickHook procedure, it is called before returning.

```
PROCEDURE SetDrawProc(theDialog: dialogPtr; theUserItem: integer; theProc:
ProcPtr);
```

SetDrawProc is a general purpose utility for telling the dialog manager which routine to call to draw your user items. It can be used with all dialogs, whether initialized with adInit or not. Call it once for each user item (except list items and gray lines in dialogs which are set up by adInit, since adInit correctly handles such items).

```
PROCEDURE SetClickHook(theDialog: dialogPtr; theHook: ProcPtr);
```

This routine tells autoDialog which routine to call when the mouse is pressed (or rather, is to be treated as having been pressed) in the given dialog, if the action required is beyond that provided by the adFilter routine. Your clickHook routine should have a case selection for each item which requires such additional processing. Do nothing for any other items. Remember that calls to setCtlValue are made automatically for radio buttons and check boxes, and that adFilter handles radio icons and lists automatically as well.

SetClickHook can be called as many times as you like. A click Hook can be removed by passing NIL for theHook.

The procedure whose pointer you pass in theHook should be declared as follows:

```
PROCEDURE myClick(theDialog: dialogPtr; VAR theEvent: eventRecord; VAR
theItem: integer; VAR Filtered: BOOLEAN): BOOLEAN;
```

For each item in the dialog, whenever the event manager reports a click in the item, adFilter will first perform the standard services, as described under adFilter, and then, if you set such a clickHook, adFilter will call your ClickHook.

The value you return for the parameter Filtered will be passed back to the dialog manager as the value of adFilter. Thus if you don't want the dialog manager to do any more processing, return TRUE.

```
PROCEDURE SetKeyHook(theDialog: dialogPtr; theHook: ProcPtr);
```

This routine tells autoDialog which routine to call when a key is pressed, if the action required is beyond that provided by the adFilter routine.

SetKeyHook can be called as many times as you like. A KeyHook can be removed by passing NIL for theHook.

The procedure whose pointer you pass in theHook should be declared exactly as for myClick, above.

Whenever the event manager reports a keyDown or autoKey event, adFilter will first perform the standard services, as described under adFilter, and then, if you have set one, adFilter will call your keyHook.

The value you return in Filtered will be passed back to the dialog manager as the value of adFilter. Thus if you don't want the dialog manager to do any more processing, return TRUE.

```
PROCEDURE getResNames(theParam: resType; VAR theIndex: integer; VAR theString: str255);
```

This is a general purpose routine suitable for building list items in adInit. If one or both of the procedures passed in the fetchProcs parameter is set to @getResNames, the corresponding list will be constructed from the non-null resource name strings of type theParam. See the description of adInit.

```
PROCEDURE dimItems(theDialog: dialogPtr; onItems, offItems: adSet);
```

This is a general purpose utility routine. Control items in onItems are given hiliting 0, those in offItems, 255. Radio icon items are greyed or not in imitation of the Control Manager's Hilite technique. No other items are affected. You may find dimItems useful in writing your clickHook or keyHook procedures.

```
PROCEDURE adGetText(theDialog: dialogPtr; theItem: integer; VAR theString: str255);
```

This is a general purpose utility routine, and simply combines calls to getDItem and getIttext. It can be used with all dialogs, whether initialized with adInit or not.

```
PROCEDURE CenterDialog(VAR wher: point; theID: integer; SetWher: Boolean);
```

Reads the dialog resource to get the bounds of the window, and centers it on the screen of the machine running it. If SetWher is FALSE, the dialog template is actually modified, and wher is undefined. If SetWher is TRUE, to satisfy SFGetFile and SFPutFile wher is returned set to the correct displacement vector of the top left corner. Call this before GetNewDialog or adInit.

```
PROCEDURE CenterAlert(theID: integer; moveVert: BOOLEAN);
```

Reads the alert resource to get the bounds of the window, and centers it on the screen of the machine running it. Call this before calling Alert, StopAlert, NoteAlert, or CautionAlert. If moveVert is FALSE, CenterAlert will use the existing top and bottom vertical coordinates in the alert template. If moveVert is TRUE, the alert is placed with twice as much space between its bottom and the bottom of the screen as betwvwn its top and the menu bar.

```
PROCEDURE adEmptySet (VAR theSet: adSet);
```

Returns an empty adSet.

```
FUNCTION adGetSelString(theDialog: dialogPtr; theItem: integer; VAR theString: str255): BOOLEAN;
```

If the given item is a list item and the user has selected something in it, adGetSelString returns TRUE, and sets theString to the first (or only) selected item. Otherwise FALSE is returned and theString is a null string.

```
PROCEDURE adDelString(theDialog: dialogPtr; theItem: integer; theText: Ptr; theLength: integer);
```

If the indicated item is a list item, and if the text pointed to is a line in that list, then that line is removed from the list. Otherwise adDelString does nothing. **WARNING:** if the text is in a relocatable block, that block should be locked!

```
PROCEDURE adAddString(theDialog: dialogPtr; theItem: integer; theText: Ptr; theLength: integer);
```

This adds the specified text as a new line in the list. **WARNING:** if the text is in a relocatable block, that block should be locked!