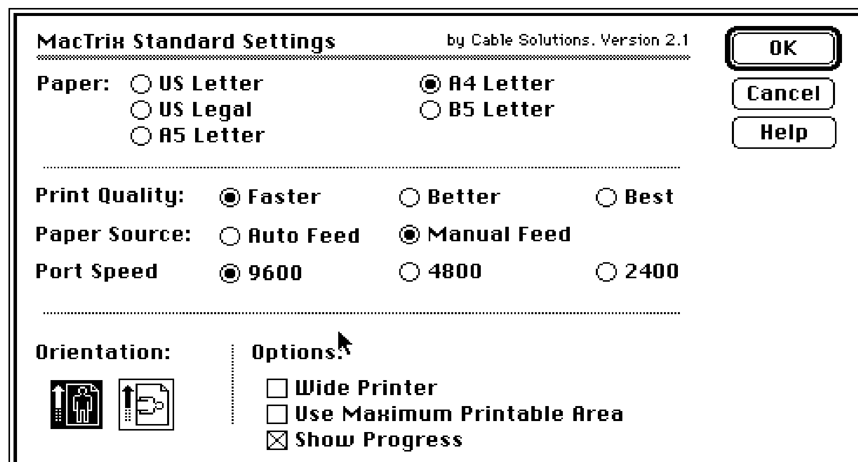
USING DIALOGS WITH OPEN PROLOG

One of the great things about the Macintosh is that you can use the mouse to select menus, click buttons and so on – you don't always have to be typing. Likewise, a lot of information can be given to you without any effort on your part as a user. As a programmer, you need to design how your program's users should interact with it, and then you need to build the facilities needed (called the program's *user interface)* into your program. Many otherwise excellent programs are spoiled by awful user interfaces; users find them difficult to use, don't learn all their features and don't get to like them. Some very ordinary programs have really terrific user interfaces; users love them, use them heavily and want more features all the time. The Macintosh has been in the forefront of good, consistent, empowering, user interface design since it was introduced. If you want good advice on user interface design, read Apple's *Human Interface Guidelines.*

To enable Open Prolog programmers to build reasonably good user interfaces, Open Prolog, starting with version 1.0.3, offers an easy-to-use toolbox for building *dialog boxes.* A Dialog Box is a window that may contain different *dialog items* like buttons, text and so on. Here is an example:



This dialog box contains buttons – e.g. OK, Cancel, Help*;Radio Buttons - e.g.*US Letter, A4 Letter*, etc.; Icons (for orientation); Check Boxes – e.g.* Wide Printer*; pictures (the dotted lines) and Static Text - e.g.* MacTrix Standard Settings. *The OK button is outlined because it is the default button, so that if the user presses the Return or Enter key, it will be clicked.*

The facilities built into Open Prolog for dialog box interfaces are for a situation where the programmer will use a tool like ResEdit to build the dialogs and will then 'drive' the dialogs using Open Prolog code. This driving of the dialogs will include responding to the user's actions and possibly modifying some of the dialog's properties, for example the text it contains. To use a dialog box, a program displays it and monitors the user's

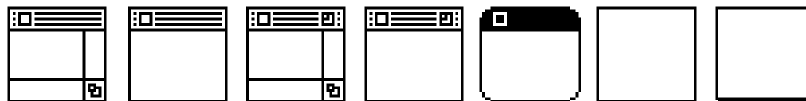
use of it. A dialog box can be *modal* or *modeless* (i.e. non-modal). If a dialog box is

modal, it is placed on top of every other window, and it won't go away until the user responds to it. Examples of modal dialogs include the Page Setup… and Print… dialogs - you must OK or Cancel them before you can continue. Apple's Human Interface Guidelines say that you should avoid modal dialogs as much as you can, because they interfere with the user's freedom of action to do what he or she wants. By contrast, a modeless, dialog box does not require the user to respond to it immediately; he or she can ignore it and do something else, maybe using the dialog later. Open Prolog allows you to build either modal or modeless dialogs into your programs.

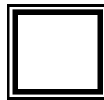
Designing Dialog Boxes

Dialog boxes are obviously highly graphical, so Open Prolog, being text-oriented, would be absolutely terrible for designing them. Instead, you need to use a program like ResEdit, Apple Computer's graphical resource design tool. ResEdit is a powerful program that allows programmers to manipulate 'resources' in Macintosh files; it is owned by Apple Computer, and is widely available at very low cost. We assume version 2.1 or later. The resources that are important to us are Dialog description resources having the code 'DLOG' (for DiaLOG), and Dialog Item description resources having the code 'DITL' (for Dialog ITem List). ResEdit enables you to design dialogs visually, on screen, containing the items you wish, having the locations and sizes you prefer. The designs are saved in files as DLOG and DITL resources.

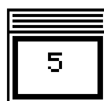
There are a number of different styles of window you can have for dialog boxes. The dialog's window style determines whether the dialog is modal, modeless or 'movable modal', in conformity with the Human Interface Guidelines. Using small pictures from ResEdit, here is a summary of the window styles and their modalities:



These window styles , with or without close boxes, are all for modeless dialogs. The rightmost two dialogs cannot be moved by the user. Grow Boxes (in the first and third boxes) are not supported.



This is the window style for modal dialogs.



This is the window style for movable modal dialogs.

ResEdit allows you to specify different kinds of items, as shown in the following picture:

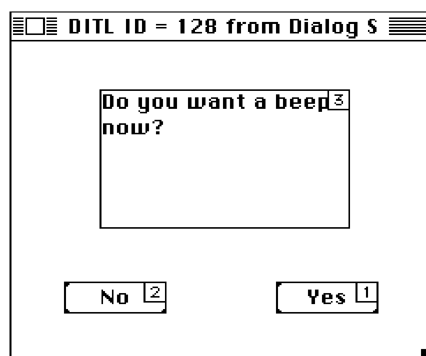
The palette of item types available in ResEdit. Open Prolog supports all types of items except user items and controls.

A Really Simple Example

Suppose you want Open Prolog put up a modal dialog to ask the user if he or she wishes to sound a beep, and then to disappear. Here is a picture of the dialog, constructed with ResEdit:

This very simple modal dialog contains just three items, two buttons and a piece of static text.

Here is a picture of the dialog item list (the DITL) for the dialog (from ResEdit):

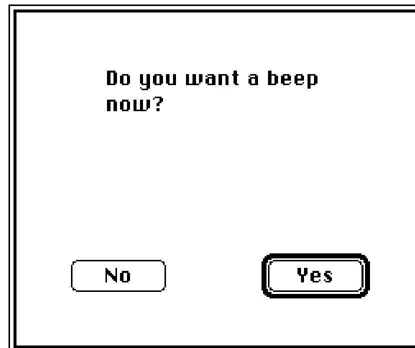


The DITL for the dialog above. Notice that each item is numbered.; Open Prolog refers to the items by that number

To use this dialog, we need to display it and then respond to the user's interactions (called *user events).* Here is a simple procedure to display this dialog, wait for the user to click a button, sound the beep if requested and finally dispose of the dialog.

```
simple_modal_dialog :-
        purge_events(dialog),
        new_dialog('Dialog Samples',128,Reference),
        wait_for_event(dialog,[Reference,Item]),
        (Item==1 /* 1 is the OK button */ -> beep;true),
        close_dialog(Reference).
```

When this procedure is executed, this is what appears on the screen:



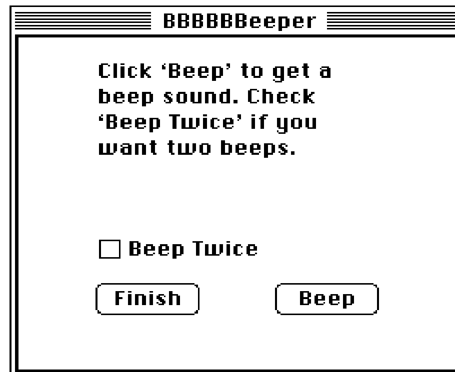
The dialog as displayed by Open Prolog

Notice that the ‘Yes’ button has a thick black line around it. This is to show that it is the ‘default’ button – typing the Return or Enter key will have the same effect as clicking it. Open Prolog will do this automatically for the first item in the dialog item list, as long as it’s a button. Likewise, if the second item is a button, typing the Escape key or using the Command-Dot combination (i.e. holding down the Command key and typing the ‘.’ key) has the same effect as clicking it. In this example, therefore, typing Return or Enter has the same effect as clicking the ‘Yes’ button, and typing the Escape key or typing Command-Dot has the same effect as clicking the ‘No’ button.

The predicates `new_dialog/3`, `purge_events/1`, `wait_for_event/2` and `close_dialog/1` are part of Open Prolog’s user interface toolkit. What happens is that `new_dialog` looks into the file ‘Dialog Samples’ for a DLOG number 128 and loads it as a dialog. It returns a *dialog reference*. Then, `wait_for_dialog_hit` keeps looking for a dialog user event. A dialog user event occurs whenever a dialog item that is *enabled* is used. In this dialog, two items are enabled - the OK button and the Cancel button. (You can specify whether an item is enabled or not using ResEdit.) Thus, in this example, whenever one of the buttons is clicked, a dialog user event occurs. When the dialog user event occurs, a description of the item involved is returned. The description is a unique specification of the item – a list comprising the dialog reference followed by the item number. For example, if the dialog reference was `6` and the Cancel button (item 2) was clicked, `wait_for_dialog_hit` would return `[6,2]`. In this example, the code ignores the dialog description because the dialog is modal, and no other dialog could be active; the code examines the item number that is returned – if it is the ‘OK’ button the beep is sounded – and the dialog is removed.

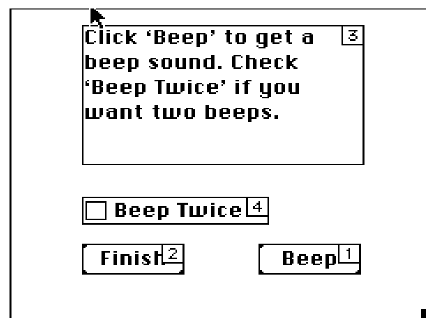
As you can see, using a simple dialog could hardly be more straightforward. Here is a slightly more complicated example, containing a check box and a slightly different functionality for the buttons.

First, the dialog, as produced using ResEdit:



This is a movable modal dialog containing four items - two buttons, a piece of static text and a check box.

Next, here's a picture of the dialog items, from ResEdit:



Next, here's the code for driving this dialog box:

```
smarter_modal_dialog :-
        purge_events(dialog),
        new_dialog('Dialog Samples','Smart Dialog',Reference),
        wait_for_event(dialog,[Reference,Item]),
        process_smarter_dialog_hits(Item,Reference),
        close_dialog(Reference).

process_smarter_dialog_hits(1,Reference) :- %1 is the Beep button
        %disable controls while beeping…
        set_dialog_property([Reference,1],active(false)),
        set_dialog_property([Reference,2],active(false)),
        set_dialog_property([Reference,4],active(false)),
        beep, %give one beep anyway
        current_dialog_property([Reference,4],value(X)), %4 is the check box
        (X==1->beep;true),
        %re-enable the controls…
        set_dialog_property([Reference,1],active(true)),
        set_dialog_property([Reference,2],active(true)),
        set_dialog_property([Reference,4],active(true)),
        wait_for_event(dialog,[Reference,Item]),!,
        process_smarter_dialog_hits(Item,Reference).
process_smarter_dialog_hits(2,Reference). %2 is the Cancel button
process_smarter_dialog_hits(4,Reference) :- %4 is the Check Box button
        toggle([Reference,4]),
        wait_for_event(dialog,[Reference,Item]),!,
        process_smarter_dialog_hits(Item,Reference).

toggle(Ref) :-
        current_dialog_property(Ref,value(V)),
```

```
            toggle_value(V,I),
            set_dialog_property(Ref,value(I)).
```

```
    toggle_value(0,1).
    toggle_value(1,0).
```

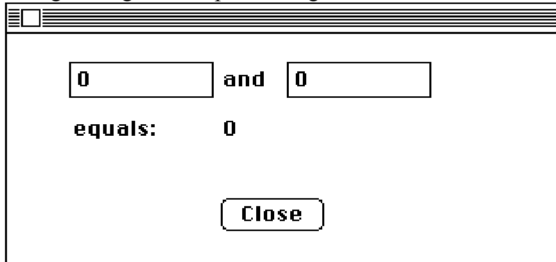
The dialog is driven by `smarter_modal_dialog`. This gets the new dialog (this time referencing it by name rather than index number), waits for a user event and then passes the event and the dialog reference to `process_smarter_dialog_hits`.

The smarter dialog as it appears when used by Open Prolog. The check box has been clicked.

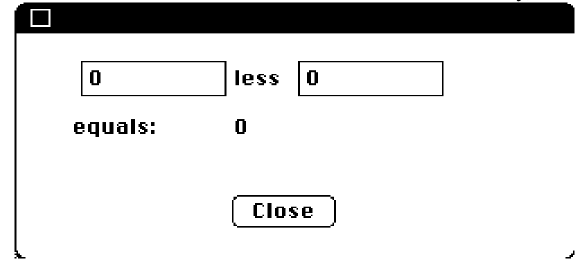
Depending on the item number, `process_smarter_dialog_hits` generates appropriate actions as follows:

- if item 1 (the 'Beep' button) is clicked, the buttons are deactivated, a beep is sounded, and, if the check box (item 4) is checked (i.e. its value is 1) then an extra beep is sounded before re-activating the buttons and waiting for another event to process and continuing;
- if item 2 (the 'Finish' button) is clicked, `process_smarter_dialog_hits` returns to `smarter_modal_dialog`, which just closes the dialog down;
- if item 4 (the Check Box) is clicked, then its state is toggled - that is, if it was already checked then it is unchecked, and if it was already unchecked then it is checked. Check boxes have the property of having a value of 1 if they are checked and 0 if they are unchecked, and you can get and set that property using the `current_dialog_property/2` and `set_dialog_property/2` predicates. (Dialogs and items have many other properties you can get and set. See later for more details.)

You can extend this idea to handle multiple dialogs simultaneously. If you do, then all the dialogs will all have to be modeless, or they will all have to be a mixture of modal and movable modal. Although using modal and movable modal dialogs is easier from a programmer's standpoint, you should really try to avoid them, because they force users to do things in a particular way; this may not be the way that suits the user best. The next example manipulates two simple modeless dialogs, one that adds the two numbers supplied, the other that subtracts them. Here are the two dialogs from ResEdit:

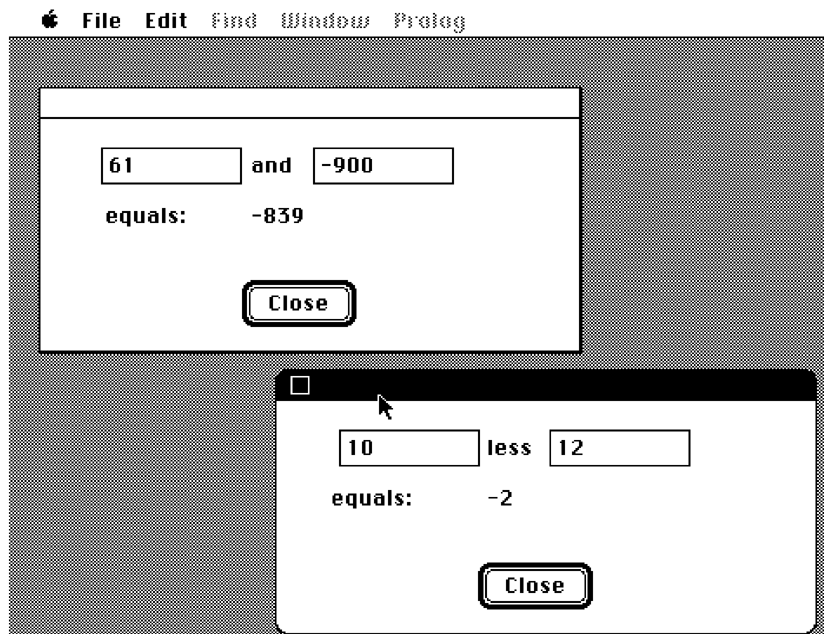


The addition dialog box.



The subtraction dialog box

Here are the dialogs as they appear when used by Open Prolog:



The two modelesss dialogs. The File menu is activated because the dialogs have close boxes, and the Edit menu is enabled because the dialogs have editable text.

Here is the code to drive both dialogs:

```
modeless_dialogs :-
        purge_events(dialog),
        new_dialog('Dialog Samples','Plus',PlusRef),
        new_dialog('Dialog Samples','Minus',MinusRef),
        wait_for_event(dialog,[Reference,Item]),
        process_modeless_dialog_hits(Item,Reference,PlusRef,MinusRef),
        close_dialog(MinusRef),
        close_dialog(PlusRef).

process_modeless_dialog_hits(1,_,_,_) :- !. % Item 1 - the 'Close' button
process_modeless_dialog_hits(1000,_,_,_) :- !. % Close Box or Close Menu
process_modeless_dialog_hits(_,Ref,PlusRef,MinusRef) :- %all other items
        current_dialog_property([Ref,2],value(N1)),
        current_dialog_property([Ref,4],value(N2)),
        (number(N1),number(N2)->
        (Ref==PlusRef->V is N1+N2;Ref==MinusRef->V is N1-N2);
        V = 'value!'),
        set_dialog_property([Ref,6],value(V)),
        wait_for_event(dialog,[Reference,Item]),
        process_modeless_dialog_hits(Item,Reference,PlusRef,MinusRef).
```

You can see that all the examples above have the same basic layout - open the dialog(s), wait for an event, process it, wait for an event, process it, and so on, until the dialog(s) are to be closed. This is fine for many situations, and is perfect for modal dialogs, but it can be awkward if you want the program to do something else while it's waiting for user events. For this reason, Open Prolog offers a completely different alternative, based on the idea of *interrupts*. Every time a dialog user event occurs, it generates an *interrupt request.* This request has a certain priority (for dialogs, the priority is 2), and if it is higher than the priority of the query being evaluated at the time, then that query is suspended and the interrupt request is serviced by looking for a specific query to evaluate. This new query (called an *interrupt handler*) must be a clause with the head:

```
handle_dialog_interrupt(Specification)
```

Where Specification is the dialog item description for the dialog and item that generated the user event. If the query can be found, it is evaluated; afterwards, the suspended query is resumed. Thus, your program could be performing its normal normal tasks, and it would be interrupted as necessary by interrupt requests coming in from user events. In other words, the program being interrupted would not have to be designed with whatever the interrupts do in mind - in fact, it would not necessarily 'know' that it was being interrupted at all. Handling events using interrupt signals is very widely used in computers, and it's called *interrupt-driven Event Handling.* It allows the ordinary programs and the event handler programs to be completely independent of each other. Furthermore, the ordinary programs can be doing something useful in the absence of interrupts, so the computer's power is not wasted as it is in all our previous examples while the computer was waiting for a user event. Thinking of it another way, you can have a query that's only called when a dialog user event occurs - it comes to life, executes and then dies.

Here is a very simple modeless dialog, using interrupt-driven Event Handling. It can be used when any program is running - a dialog event causes an interrupt which suspends the program, performs the event handling query, and then allows the program to resume. Here's the dialog itself:

A very simple modeless dialog, as it appears from Open Prolog

There are two separate pieces of code. First, the code that opens up the dialog:

```
modeless_beep_dialog :-
       new_dialog('Dialog Samples','Modeless Beep',_).
```

As you can see, all it does is open the dialog - it doesn't process any events. The code for handling events is installed as a dialog interrupt handler, as follows:

```
handle_dialog_interrupt([D|Spec]) :-
       current_dialog_property([D],name('Beeper')),
       process_modeless_beep_dialog([D|Spec]).
```

```
process_modeless_beep_dialog([_,1]) :- beep.
```

```
process_modeless_beep_dialog([D,1000]) :- close_dialog(D).
```

The dialog interrupt handler firstly checks that the event is for our dialog. This is done by ensuring that the name of the dialog in the interrupt is the same as the name[1] of our dialog. If so, then our process_modeless_beep_dialog is called to handle the two simple cases.

When you are building the handler for an interrupt-driven dialog, there is one rule you must follow so that your handler lives in harmony with the rest of the system. It is possible, indeed it is likely, that your handler will be called in response to ‘foreign’ dialog events, that is, dialog user events for other dialogs that may be active in Open Prolog. To allow foreign dialog events to be processed by their own handlers, your handler must fail when if is invoked with them. If your handler succeeds when called in response to foreign dialog user events, then their handlers will never see them and so their dialogs will stop working. So, the rule is: *your dialog interrupt handler must fail for foreign dialog user events.*

Summary of Information

Specifications

A *specification*, which is always a list, is used to denote either the dialog toolkit itself, a dialog, a dialog item, or part of a dialog item.

For example, [] denotes the dialog toolkit itself, [2] denotes dialog reference 2, [2,4] means item 4 of dialog 2, [2,4,8] means sub-item 8 of item 4 of dialog 2. Standard items have no sub-items, but menus and lists will have.

Properties

You can get and set the *properties* of dialogs and dialog items with current_dialog_property/2 and set_dialog_property/2 respectively. The first argument is the specification of the entity whose property you wish to process; the second item is the property itself.

Here is a list of all the properties understood by the dialog toolkit at present:

```
components(R) %number of subsidiary parts
kind(Kind) %the kind of the entity
name_list(Na) %entity's name as an ascii list
value_list %entity's value as an ascii list
visible(Vi) %boolean - visibility of the entity
enabled(E) %boolean - enabled to respond to clicks
rect(R) %enclosing rectangle of entity [Top,Left,Bottom,Right]
indexed_component(I,D) %returns reference number of the Ith item
closeable(C) %boolean - can be closed
font(F) %name of font for the entity
font_size %font size of font for the entity. 0 means default
value(V) %value of the entity
```

[1]All dialogs can have names, though in some dialog types the names can't be seen by the user. See ResEdit.

```
name(Va) %name of the entity
active %boolean - activity status (i.e. black or gray),
category %general category of the entity
true_rect(R) %the true, undistorted size of the entity (e.g. a picture)
front(Boolean) %boolean - the dialog window specified is the front window
front_window(F) %the dialog reference of the front window (if it exists)
exists(Existence) %boolean - does the entity specified exist
```

Note that not all entities will have all properties. If you try to get such a property, the predicate should silently fail; if you try to set such a property, it may fail or complain.

Dialog Kinds are:

```
documentBoxWithGrow
dialogBox
plainDialogBox
alternateDialogBox
documentBox
movableModalDialogBox
documentBoxWithZoomAndGrow
documentBoxWithZoom
roundedRectDocumentBox
```

Dialog Item Kinds are:

```
button
checkBox
radioButton
staticText
editText
icon
picture
defaultButtonOutline
```