Quick Introduction to Menubars and Menus

The classes defined in the Be Interface Kit for implementing menus and menubars are powerful, but more flexible than are needed for most applications. This document is aimed at getting you up and running quickly with menus and menubars, if what you desire is a standard menubar across the top of a window, with normal menu items, submenus, radio menu entries, and so forth.

The classes of interest when creating menu bars are **`BMenuBar`**, **`BMenu`**, and **`BMenuItem`**, which respectively represent menubars, menus within menubars, and menu items within menus. A fourth class, `BMenuField`, is used with popup menus, and is not discussed here. Remember that all of these classes provide considerably more functionality than is discussed here; refer to the Be reference documentation if you need more power in your menus.

Note: Some errors may have crept into some of the code shown here, when it was being formatted for readability. However, the the full program (which is given at the end of this document) contains all of the code shown, and has been verified to compile and work correctly, by transferring it back to the compiler after formatting, and using it!

Creating a Menubar and Menus

Menubars are created with the **`BMenuBar()`** constructor. They inherit from the **`BView`** class, and like all `BView` objects, are inserted into a parent window using that window's **`AddChild()`** function. Menus are then created with the **`BMenu()`** constructor, and are added to a parent menubar using that menubar's **`AddItem()`** function.

Let's look at this with a simple example. The following code will create a window containing a menubar and two empty menus. Menu-specific code is shown in bold; the remainder of the code has to do with defining the necessary application and window objects to actually run the application.

```
#include <Application.h>
#include <InterfaceKit.h>

const char *APP_SIGNATURE= "application/x-vnd.Be-MyMenus";

class MyMenusWindow : public BWindow {
   public:
      MyMenusWindow::MyMenusWindow(BRect frame)
         : BWindow(frame, "Hello World", B_TITLED_WINDOW, B_NOT_ZOOMABLE)
{
         /* Make a menu bar and add it to the window. "mbarFrame"
            is passed into the BMenuBar constructor only to set the
            coordinates of the top left corner of the menubar in
```

```
            the window; the bottom and right coordinates are, by
            default, ignored. */
         BRect mbarFrame(0, 0, 0, 0);
         BMenuBar *mbar = new BMenuBar(mbarFrame, "MyMenuBar");
         AddChild(mbar);

         /* Create and add two menus to the menubar */
         BMenu *menu1, *menu2;
         menu1 = new BMenu("Hello");
         mbar->AddItem(menu1);
         menu2 = new BMenu("Goodbye");
         mbar->AddItem(menu2);

         /* Show the window. */
         Show();
      }

      /* QuitRequested causes clicks on the window's "close"
         button in the titlebar to quit the application. Without
         this, we'd have to kill the application manually. */
      bool MyMenusWindow::QuitRequested() {
         be_app->PostMessage(B_QUIT_REQUESTED);
         return true;
      }
};

class MyMenusApp : public BApplication {
   public:
      MyMenusApp::MyMenusApp() : BApplication(APP_SIGNATURE) {
         BRect windowRect;
         windowRect.Set(50,50,349,399);
         new MyMenusWindow(windowRect);
      }
   private:
      MyMenusWindow*theWindow;
};

/* The "main" function creates and runs the application.*/
void main(void) {
   MyMenusApp *theApp;
   theApp = new(MyMenusApp);
   theApp->Run();
   delete theApp;
}
```

The **`BMenuBar()`** constructor is used with two arguments. The first is a `BRect` which defines the location of the top left corner of the menu bar within its parent window; for standard menubars, this is (0, 0). Note that the bottom and left corners of the `BRect` are ignored by the default form of `BMenuBar()`, which is used here. The second argument to `BMenuBar()` is a string to provide a name for the menu bar. Once created, the menubar is added to an appropriate window using that window's `AddChild()` function.

Once a menubar has been created, new menus for it are created with the **`BMenu()`** constructor, and inserted into the menubar using the **`BMenuBar::AddItem()`** method. Menus inserted first will show up on the left side of the menubar, menus inserted last will show up on the right side.

Creating Menu Items in a Menu

Let's put a couple of menu items into the previously created menus. This requires a few more lines of code, inserted immediately after the menu-relevant (bolded) code in the example above:

```
/* Create and add some items to the first menu */
BMenuItem *menu1item1 = new BMenuItem("Bonjour", NULL);
menu1->AddItem(menu1item1);
menu1->AddItem(new BMenuItem("G'day Mate", NULL));
/* Create and add some items to the second menu. */
menu2->AddItem(new BMenuItem("Au Revoir", NULL));
menu2->AddItem(new BMenuItem("Auf Wiedersehen", NULL));
menu2->AddItem(new BMenuItem("Sayonora", NULL));
```

New menu items are created via a call to the **`BMenuItem()`** constructor, which in its simplest form takes two arguments. The first argument to `BMenuItem()` is the string which is displayed onscreen for that item in the menu, and the second argument is a `BMessage` which is sent to the application when the menu item is invoked by the user[1]. We'll discuss how menus communicate with the application a bit later on, so for now we'll use a null pointer as the second argument to `BMenuItem()`; this means our menu items will do nothing when selected.

Once created, menu items are added to an appropriate menu using that menu's **`AddItem()`** function. Items inserted first appear at the top of the menu, items inserted last appear at the bottom. You might note that both the `BMenu` and `BMenuBar` classes have an `AddItem()` function; this isn't surprising, because `BMenuBar` is actually a subclass of `BMenu`. Loosely speaking, a `BMenuBar` object is just a `BMenu` object which is always visible and which is displayed horizontally instead of vertically.

Shortcut Keys for Menu Items

A shortcut is a key combination, unique within the window, which can be used to invoke a menu item without actually clicking on the menu and the menu item. A common example is the **Alt-C** key combination which, in most applications, is the shortcut bound to the **Copy** menu item in the **Edit** menu. Shortcuts are associated with

1. Actually the `BMessage` we pass in here is never sent to any other object. Instead, it serves as a *model message*, and whenever the menu item needs to send a message, it copies the model message, adds some additional information, and sends that modified copy. This permits the receiver of the message to safely delete the message, without affecting the original in the `BMenuItem`.

a menu item via optional third and fourth arguments to the `BMenuItem()` constructor. The third argument, if present, defines the key to press in conjunction with the **Alt** key, in order to invoke the shortcut; in the **Alt-C** shortcut example, this argument would be the character `'C'`. The fourth argument, if present, defines additional modifier keys which must be pressed along with the **Alt** key, to invoke the argument; these can be the **Shift**, **Control**, or **Option** keys. Note that the **Alt** key must always be pressed to invoke a menu shortcut.

Let's add shortcuts to the two menu items we created above. Change the code for them to the following:

```
/* Create and add some items to the first menu */
menu1->AddItem(new BMenuItem("Bonjour", NULL, 'B'));
menu1->AddItem(new BMenuItem("G'Day Mate", NULL, 'G',
                B_SHIFT_KEY|B_CONTROL_KEY|B_OPTION_KEY));
```

When you compile and run the program, you'll now see that the **Bonjour** menu item appears with the shortcut **Alt-B** beside it, and the **G'Day Mate** item appears with the shortcut **Shft-Ctl-Opt-Alt-G** beside it. Pressing these two key combinations on the keyboard will now invoke the corresponding menu items.

The fourth argument to `BMenuItem()`, when present, is a bit mask constructed by ORing any or all of the constants **`B_SHIFT_KEY`**, **`B_CONTROL_KEY`**, and **`B_OPTION_KEY`** together. All of them are used in the example above.

The case of the character passed in as the third argument to `BMenuItem()` is preserved when displaying the shortcut in the menu (i.e. `'b'` is displayed lower-case, `'B'` is displayed upper-case), but is irrelevant in terms of the operation of the shortcut; **Alt-b** and **Alt-B** both mean the same thing. To avoid confusion on the part of your users, it is recommended that you always use upper-case characters.

Submenus

A submenu is a menu that is accessed via a menu item in a "higher-level" menu. Just as a menu item is added via a line of code something like this:

```
menu->AddItem(bMenuItemPtr);
```

where `bMenuItemPtr` is a pointer to an object returned by the `BMenuItem` constructor, so a submenu is added like this:

```
menu->AddItem(bMenuPtr);
```

where `bMenuPtr` is a pointer returned by the `BMenu()` constructor we've already used in setting up our top-level menus.

To see this in action, add the following code immediately after the two `menu1->AddItem(...)` invocations in the current code:

```
/* Create a submenu. */
BMenu *subMenu = new BMenu("A Submenu");
subMenu->AddItem(new BMenuItem("Subitem 1", NULL));
subMenu->AddItem(new BMenuItem("Subitem 2", NULL));

/* Add the submenu to the bottom of the first toplevel menu.*/
menu1->AddItem(subMenu);
```

Checked and Unchecked Menu Items

You may sometimes wish to mark a menu item as selected, by a check mark next to it; for example, a menu item called **Show Special Characters** might be checked in a word processor to indicate that the word processor should show special characters (such as linefeeds) visually on the screen, and be unchecked if such characters should not be shown. Typically the user is able to toggle such an option off or on by selecting it. The Be menu classes allow you to mark or unmark a menu item, but do not automatically handle toggling it–you'll have to do that manually in your application, something we'll get to a bit later.

Marking an item is done with the **`BMenuItem::SetMarked()`** function, which takes a single `bool` argument, and marks the item if the boolean value is true, and unmarks it if the boolean value is false. To create a new, initially checked item in the first menu, add the following code immediately after the code added in the section just above:

```
/* Create a new, checked item. */
BMenuItem *checkedItem = new BMenuItem("Toggle Me", NULL);
checkedItem->SetMarked(true);
menu1->AddItem(checkedItem);
```

If you run the application after compiling it with this addition, you'll find that the new item does show up checked, but that selecting it does not make it unchecked; this must be done with internal code, something we'll get to later.

Radio Menus

A radio menu is one in which only one item can be marked with a check mark; selecting another item in that menu causes the previously marked item to be unmarked, and places a check beside the newly marked item. The `BMenu` class has built-in support for radio menus; by invoking the **`BMenu::SetRadioMode()`** function on a menu, you can indicate that it should function as a radio menu, and `BMenu` will take care of most of the details. To see this in action, let's add a radio submenu to our first top-level menu, as follows (code should be inserted immediately after that given just above):

```
/* Create a new radio submenu.*/
BMenu *radioMenu = new BMenu("Choose One");
/* We'll need to call SetMarked on the first radio item to mark it,
   so keep a ref to it in a variable. */
```

```
        BMenuItem *radioItem1 = new BMenuItem("Choice 1", NULL);
        radioMenu->AddItem(radioItem1);
        /* Mark the first radio item as checked--BMenu doesn't do this automatically.
*/
        radioItem1->SetMarked(true);
        radioMenu->AddItem(new BMenuItem("Choice 2", NULL));
        radioMenu->AddItem(new BMenuItem("Choice 3", NULL));
        radioMenu->SetRadioMode(true);
        /* Add the radio menu as a submenu of menu1. */
        menu1->AddItem(radioMenu);
```

If you now run the application, you will find a new submenu which behaves as a radio menu should. It's the invocation of `SetRadioMode()` with a `true` argument which caused the new menu to be treated as a radio menu; naturally, invoking `SetRadioMode()` with a `false` argument would make the menu a normal, non-radio menu. Note that putting a menu into radio mode does not automatically cause one of its items to be checked; you must use `BMenuItem::SetMarked()` to specify the initially marked item, as was done above. (It is possible to start with no items marked in a radio menu, but this is not what is usually desired.)

Finding Out What is Marked

To use a radio menu in your application, you need some way of determining which element of the radio menu a user has chosen. As is discussed later, each single menu item in your menu hierarchy will send a `BMessage` to the containing window, when that menu item is selected by the user, and the simplest strategy with radio menus is to have each item in a radio menu send a uniquely identifiable `BMessage`. The window receiving these messages can then determine which radio menu item has been selected (and is therefore now marked), and take an appropriate action.

An alternative is to have each item in the radio menu send the same `BMessage` (basically just telling the parent window, "Something in this radio menu has been selected".) The window message-handling code can then extract the radio menu from that `BMessage`, and use the **`BMenu::FindMarked()`** function to discover which item is marked, something along these lines;

```
/* myRadioMenu is a BMenu configured as a radio menu. */
BMenuItem *markedItem;
markedItem = myRadioMenu->FindMarked();
```

If there is any marked item in `myRadioMenu`, `markedItem` will now contain a pointer to that item's associated `BMenuItem` object. If there is no marked item, `markedItem` will contain `NULL`.

Menu Separators

A menu separator is a horizontal line which separates one part of a menu from another. Separators are used to visually divide a menu into related groups of items, and have no function other than aesthetics. To add a separator to the end of an existing menu, just invoke

```
menu->AddSeparatorItem();
```

and then continue using `AddItem()` with the menu to add items after the separator. I don't really need to show an example of this, do I?

Disabling and Enabling Menus and Menu Items

When you first create a menu (or a menu bar), the items in it are enabled, which means they may be meaningfully selected by the user, and are displayed as black text. You may disable menu items or entire menus (or enable previously disabled items) using the **`BMenuItem::SetEnabled()`** or **`BMenu::SetEnabled()`** functions, which are passed a single `bool` argument—true if you want the item enabled, or false if you want it disabled. When a menu or a menu item is disabled, its text is "grayed out", and selecting it will not cause any internal action to be performed (i.e. a disabled menu item will never send a message to its associated window object.) If the disabled item is a menu (i.e. a top-level menu in the menubar, or a submenu), then any other items accessible through it are also disabled, and remain disabled until you enable the parent menu again. However, the user can still "browse" these disabled items, even though selecting them will not do anything

To see this in action, look for the line `menu1->AddItem(subMenu);` in the code above, and immediately after that line, add the following:

```
/* Disable this subMenu (and everything in it). */
subMenu->SetEnabled(false);
```

This code will cause the `subMenu` item of `menu1` to become disabled. Since this item is actually a submenu, all entries within that submenu will also be disabled. To re-enable it, just execute `subMenu->SetEnabled(true);`. Use the same function to disable/enable single (i.e. not a submenu) menu items. For example, to disable **Bonjour** in the **Hello** menu, add the following line anywhere after `menu1item1` is initialized;

```
menu1item1->SetEnabled(false);
```

`subMenu` is a `BMenu` object, and `menu1item1` is a `BMenuItem` object, but that's fine–both of these classes implement a `SetEnabled()` function. Note that you can also you can also use `SetEnabled()` to disable (or enable) an entire top-level menu, in which case the name of that menu in the menubar will grayed out.

Communicating Menu Item Selections to the Parent Window

A menu is associated with a particular window (an instance of a subclass of `BWindow`, the subclass being defined by you), and menu item invocations are communicated to that window by sending it **`BMessage`** objects associated with particular menu items. So far, we've used `NULL` in place of `BMessage` objects when calling the `BMenuItem` constructor; now we'll see how to implement message functionality.

BMessages and the 'what' Field

Each `BMessage` carries with it a **`what`** field, a 32-bit constant indicating what "kind" of message this is. In the case of `BMessage` objects sent by menu items, this is often the only data needed; you can simply have each menu item send a `BMessage` with its own unique `what` value, and examine that value in your menu-handling code, to determine what action to take. We'll define `BMessage` objects, and handling code, for two of the menu items in the **Hello** menu; the **Bonjour** item and the **Toggle Me** item. The first thing to do is to define the unique constants to be used in each menu item's associated `BMessage`. Since a 32-bit value is just four ASCII characters, the easiest thing may be to define these constants in terms of the first four characters in the menu item name, assuming this results in unique values. The following lines can be placed immediately after the `#include` directives in the code file:

```
const uint32 BONJOUR = 'Bonj';
const uint32 TOGGLE = 'Togg';
```

Of course, `'Bonj'` and `'Togg'` could be referred to directly when building the needed BMessage objects, and not defined as constants, but defining them as constants in this manner is both good programming style, and a worthwhile contribution towards internal documentation of the code.

Note: Whenever using `BMessage` objects, it is vital to ensure that the `what` values defined by yourself do not conflict with those defined internally within the Be OS. This is easy to do; Be promises to use only uppercase characters and the underbar character in such values. Since `'Bonj'` and `'Togg'` both include lowercase characters, they are safe from conflicting with internally defined `what` values.

Handling BMessages with MessageReceived()

Now that we know how to identify the messages our menus might send, we can write a **`MessageReceived()`** message-handling function in the `MyMenusWindow` class[1]. It looks like this:

```
void MessageReceived(BMessage *message) {
switch(message->what) {
```

```
case BONJOUR: {
   BPoint pt = BPoint(20, 20);
   DrawString("Hi to you too!", pt)
   break;
}
case TOGGLE: {
   /* We'll put something in here later. */
   break;
}
default: {
   /* Other messages are passed to the inherited
      MessageReceived function. */
   BWindow::MessageReceived(message);
   break;
}
```

In response to an invocation of the **Bonjour** menu item, we'll draw a string into the window, near the upper left corner of the drawing area. We won't handle the **Toggle** item quite yet–that's a bit in the future. Any other messages are passed on to the `MessageReceived()` function of the parent `BWindow` class, in case that function understands them.

The final piece of the puzzle is to associate an appropriate `BMessage` with the **Bonjour** menu item. Go back to where the **Bonjour** `BMenuItem` is created, and for the line

```
BMenuItem *menu1item1 = new BMenuItem("Bonjour", NULL);
```

substitute the lines:

```
BMessage* bonjourMessage = new BMessage(BONJOUR);
BMenuItem *menu1item1 = new BMenuItem("Bonjour", bonjourMessage);
```

In other words, where previously we'd passed in `NULL` as (the pointer to) a `BMessage`, now we are using `bonjourMessage` as the message, and that message includes the `BONJOUR` constant in its `what` field, enabling the message receiver to identify the source of the message as the **Bonjour** menu item.

Contents of BMessages from BMenuItems

Messages from menus contain more information than the `what` field. In addition to any information you might store in the message when you create it, the BeOS automatically inserts three other pieces of data into any message originating from a `BMenuItem`; '**when**', '**source**', and '**index**'. '**when**' gives the time the menu item was invoked by the user, in microseconds from the start of the year 1970. '**source**'

1. Whenever a message is sent by the Be OS to an object that can receive it, that object's `MessageReceived()` member function is invoked to handle the incoming message. Here, we are overriding `BWindow::MessageReceived()` with `MyMenusWindow::MessageReceived()` to handle our menu item messages as we desire. Note in the code that messages not from our menu items are passed on to `BWindow::MessageReceived()` by `MyMenusWindow::MessageReceived()`! This ensures that messages other than those generated by our menu items are handled correctly.

contains a pointer to the invoking BMenuItem, and '**index**' gives the index of that menu item within its parent menu, starting with 0 as the first item in the parent, etc.

Using this information, let's fix the **Toggle Me** menu item so that selecting it actually toggles its checkmark on and off. First, associate an appropriate message with the toggle `BMenuItem`. In the code, look for the line

```
BMenuItem *checkedItem = new BMenuItem("Toggle Me", NULL);
```

and replace it with

```
BMessage *toggleMessage = new BMessage(TOGGLE);
BMenuItem *checkedItem = new BMenuItem("Toggle Me", toggleMessage);
```

as we did this for the **Bonjour** `BMenuItem`. (The `TOGGLE` constant was defined previously, at the same time as the `BONJOUR` constant.)

Now, in the branch of the message-handling `switch` that takes care of the `TOGGLE` case, i.e.

```
case TOGGLE: {
   /* We'll put something in here later. */
   break;
}
```

add in code so it looks like this:

```
case TOGGLE: {
   BMenuItem *toggleItem;
   /* Extract the originating menu item from the message */
   message->FindPointer("source", (void **)&toggleItem);
   /* If the item is currently marked with a check... */
   if (toggleItem->IsMarked()) {
      /* ...unmark it... */
      toggleItem->SetMarked(0);
   }
   else {
      /* ...otherwise it is not currently marked, so mark it. */
      toggleItem->SetMarked(1);
   }
   break;
}
```

The key to being able to do this is the use of the `BMessage::FindPointer()` function to extract a pointer to the invoking `BMenuItem` from the `BMessage`. The odd-looking typecast `(void **)` on `&toggleItem` is correct–we need to pass in a pointer to a pointer to a `BMenuItem`, so that `FindPointer()` can copy the `BMenuItem` pointer into our variable correctly.

Error Handling when Building Menus

It's rare for any of the menu construction functions to terminate with an error–they consume few resources, and are typically executed during a program's startup, when

the program has most of its resources free. For this reason, most menu code is written without worrying too much about error handling.

Probably the most important place to worry about error handling is when extracting data from the BMessage objects sent by your menu items. Since C++'s strong type-checking is defeated when sending data in BMessages, you might want to check to make sure there is data associated with the names you look up in a BMessage.

Of the menu-specific function discussed above, only `AddItem()` is can return values which indicate an error, and then only if a different form of `AddItem()` than that discussed above is used.

Summary

Here's the complete program that follows from the instructions given earlier in the text. Bolded code corresponds roughly to menu-related features or usages that have not been seen previously in the program. Copy this code into a single-file BeIDE project, compile, and then start playing with it to see what you can do. Have fun!

```
   #include <Application.h>
   #include <InterfaceKit.h>

   const uint32 BONJOUR = 'Bonj';
   const uint32 TOGGLE = 'Togg';

   const char *APP_SIGNATURE= "application/x-vnd.Be-MyMenus";

   class MyMenusWindow : public BWindow {
      public:
         MyMenusWindow::MyMenusWindow(BRect frame)
            : BWindow(frame, "Hello World", B_TITLED_WINDOW, B_NOT_ZOOMABLE)
{

            /* Create a view for the interior region of the window
               (everything other than the menu bar) and add it to
               the window. */
            interior = new BView(BRect(30, 30, 200, 200), "", B_FOLLOW_NONE,
B_WILL_DRAW);
            AddChild(interior);

            /* Make a menu bar and add it to the window. "mbarFrame"
               is passed into the BMenuBar constructor only to set the
               coordinates of the top left corner of the menubar in
               the window; the bottom and right coordinates are, by
               default, ignored. */
            BRect mbarFrame(0, 0, 0, 0);
            BMenuBar *mbar = new BMenuBar(mbarFrame, "MyMenuBar");
            AddChild(mbar);

            /* Create and add two menus to the menubar */
            BMenu *menu1, *menu2;
            menu1 = new BMenu("Hello");
```

```
            mbar->AddItem(menu1);
            menu2 = new BMenu("Goodbye");
            mbar->AddItem(menu2);

            /* Create and add some items to the first menu */
            BMessage* bonjourMessage = new BMessage(BONJOUR);
            BMenuItem *menuitem1 = new BMenuItem("Bonjour", bonjourMessage);
            menu1->AddItem(menuitem1);
            menu1->AddItem(new BMenuItem("G'Day Mate", NULL, 'G',
                           B_SHIFT_KEY|B_CONTROL_KEY|B_OPTION_KEY));

            /* Create a submenu. */
            BMenu *subMenu = new BMenu("A Submenu");
            subMenu->AddItem(new BMenuItem("Subitem 1", NULL));
            subMenu->AddItem(new BMenuItem("Subitem 2", NULL));
            /* Add the submenu to the bottom of the first toplevel menu.*/
            menu1->AddItem(subMenu);
            /* Disable this subMenu (and everything in it). */
            subMenu->SetEnabled(false);

            /* Create a new, checked item. */
            BMessage *toggleMessage = new BMessage(TOGGLE);
            BMenuItem *checkedItem = new BMenuItem("Toggle Me",
toggleMessage);
            checkedItem->SetMarked(true);
            menu1->AddItem(checkedItem);

            /* Create a new radio submenu.*/
            BMenu *radioMenu = new BMenu("Choose One");
            /* We'll need to call SetMarked on the first radio item to mark it,
               so keep a ref to it in a variable. */
            BMenuItem *radioItem1 = new BMenuItem("Choice 1", NULL);
            radioMenu->AddItem(radioItem1);
            /* Mark the first radio item as checked--BMenu doesn't do this
automatically. */
            radioItem1->SetMarked(true);
            radioMenu->AddItem(new BMenuItem("Choice 2", NULL));
            radioMenu->AddItem(new BMenuItem("Choice 3", NULL));
            radioMenu->SetRadioMode(true);
            /* Add the radio menu as a submenu of menu1. */
            menu1->AddItem(radioMenu);

            /* Create and add some items to the second menu. */
            menu2->AddItem(new BMenuItem("Au Revoir", NULL));
            menu2->AddItem(new BMenuItem("Auf Wiedersehen", NULL));
            menu2->AddItem(new BMenuItem("Sayonora", NULL));

            /* Show the window. */
            Show();
        }
        /* QuitRequested causes clicks on the window's "close"
           button in the titlebar to quit the application. Without
           this, we'd have to kill the application manually. */
        bool MyMenusWindow::QuitRequested() {
            be_app->PostMessage(B_QUIT_REQUESTED);
            return true;
        }
```

```
   void MessageReceived(BMessage *message) {
      switch(message->what) {
         case BONJOUR: {
            BPoint pt = BPoint(100, 100);
            interior->DrawString("Hi to you too!", pt);
            break;
         }
         case TOGGLE: {
            BMenuItem *toggleItem;
            /* Extract the originating menu item from the message */
            message->FindPointer("source", (void **)&toggleItem);
            /* If the item is currently marked with a check... */
            if (toggleItem->IsMarked()) {
               /* ...unmark it... */
               toggleItem->SetMarked(0);
            }
            else {
               /* ...otherwise it is not currently marked, so mark it. */
               toggleItem->SetMarked(1);
            }
            break;
         }
         default: {
            /* Other messages are passed to the inherited
               MessageReceived function. */
            BWindow::MessageReceived(message);
            break;
         }
      }
   }
private:
   BView *interior;
};

class MyMenusApp : public BApplication {
   public:
      MyMenusApp::MyMenusApp() : BApplication(APP_SIGNATURE) {
         BRect windowRect;
         windowRect.Set(50,50,349,399);
         new MyMenusWindow(windowRect);
      }
   private:
      MyMenusWindow*theWindow;
};

/* The "main" function creates and runs the application.*/
int main(void) {
   MyMenusApp *theApp;
   theApp = new(MyMenusApp);
   theApp->Run();
   delete theApp;
}
```