For further information, please contact us at one of the following addresses. Please note that we cannot provide assistance in Macintosh programming fundamentals or with the use of your chosen development system.

Bare Bones Software, Inc.
P.O. Box 1048
Bedford, MA 01730-1048
**phone** (781) 687-0700
**fax** (781) 687-0711
**email** `devsupport@barebones.com`
**world wide web** `http://www.barebones.com/`

Version 8.4 June 21, 2005

# Writing Plug-Ins for BBEdit and TextWrangler

## *A Note on Nomenclature*

BBEdit and TextWrangler can both run plug-ins written following the guidelines in this document. Wherever "BBEdit" is used as the name of the application hosting the plug-in, either product may actually be doing the hosting.

This document will describe how to build Mach-O bundled plug-ins, using Apple's Xcode IDE, for use with BBEdit 8.0 or later, or TextWrangler 2.0 or later.

## *Introduction*

BBEdit has a facility for calling code modules that are not part of the application itself. The main reason for this facility is so that any sufficiently motivated individual can add specific functionality to BBEdit that goes beyond BBEdit's own charter. For example, one such code module might prepend Usenet attributions to each line in a selected range of text. This is a useful function, but it's not of interest to everyone. This document explains how to construct code modules that extend BBEdit's capabilities, and how to take advantage of the services that BBEdit offers to these code modules.

# Plug-in Gestalt

BBEdit plug-ins are intended to function as "one-shot" text filters, which function with little or no user intervention. For example, an "Educate Quotes" plug-in might spin through the text in the front editing window, and convert straight quotes to curly quotes in an intelligent fashion. This sort of task is a simple transformation on the text, and requires no user action (aside from choosing the plug-in's name as a menu command). A more complicated plug-in might pose a modal dialog before performing its transformations, so that the user can specify various options which would affect the plug-in's behavior.

Naturally, there's no constraint on the nature of the transformation that's performed. Indeed, there's no requirement that an plug-in perform any transformations — of course, since BBEdit *is* a text editor, it makes sense that an plug-in should perform some operation that is remotely text-related.

In order to aid in the task at hand, BBEdit offers a rich set of callback services. These services are intended to save the plug-in writer the task of implementing some fairly tricky and/or tedious pieces of code, which often correspond to problems that have been solved inside of BBEdit itself.

With only a few exceptions, the plug-in callbacks do not provide any measure of control over BBEdit itself, nor is it possible for a plug-in to affect BBEdit's basic editing behavior.

Plug-ins are transient pieces of code: BBEdit only loads an plug-in into memory in order to call it, and once the plug-in has returned, it is released from memory. Therefore, you should not count on any global data in your plug-in being preserved between invocations of your plug-in, nor should you expect your plug-in to be called on a periodic or repeating basis.

Of course, since plug-ins are compiled code, you're free to do whatever you wish. However, bear in mind that by living within the guidelines and constraints of the plug-in system, your code module will act and feel more like a natural extension of BBEdit itself, which is the whole point behind BBEdit plug-ins.

BBEdit plug-ins should be as friendly as possible. They should take care to release any memory that they allocate while running. In the spirit of one-shot text filters, they should leave no windows on the screen after they return to BBEdit. They should put no menus in the menu bar, and should not have an event loop. (They *can* call `ModalDialog()`. It's recommended that you use, or layer on top of, the standard filter that BBEdit provides.)

# Construction of Plug-ins

Plug-ins are packaged as a CFBundle, which includes the Mach-O implementation of the plug-in, and a property list, which describes some of the module's details.

The bundle signature of a plug-in file is '`BBXT`'. The sample Xcode project supplied with this documentation is a very simple "Hello, World", which has all the structure necessary to build a more complex plug-in.

## The Info.plist

The Info.plist file contains an XML description of the plug-ins, and other information needed by BBEdit.

The top-level dictionary of the plist contains the standard CFBundle keys and values, and then proceeds with the plug-in specific data. The top-level key "com.barebones.bbxtinfo" contains an array of one or more dictionaries. Each dictionary refers to a single plug-in contained within the bundle. Each dictionary should contain the following keys and values:

| | |
|---|---|
| `BBXFDoesNotModifyEditWindow` | A Boolean value that indicates if the plug-in does not modify the active document. |
| `BBXFRequiresEditWindow` | A Boolean value that indicates whether the plug-in requires a window it can write to. |
| `BBXFRequiresSelection` | A Boolean value that indicates whether the plug-in requires a selection in the active document. |
| `BBXFSupportsRectangularSelection` | A Boolean value that indicates whether the plug-in is rectangular selection savvy. |
| `BBXFSupportsUndo` | A Boolean value, that if set, indicates that your plug-in will call PrepareUndo()/CommitUndo() as necessary around window content manipulations. |
| `BBXTHelpBookName` | A String value with the name of the help book associated with the plug-in. |
| `BBXTHelpPagePath` | A String value containing the path to the help book. |
| `BBXTOnlineInfoURL` | A String value with a URL for the developer web page. |
| `MainFunctionName` | The Plug-in's main entry point (usually main()). If using C++, declare your main() as 'extern "C"' to avoid problems caused by C++ name mangling. |
| `MenuName` | The name used in the Tools menu to represent this plug-in. |

If the plug-in is to be scriptable, there needs to be an 'aete' resource whose name matches the "`MenuName`" key. See "Writing Scriptable Plug-Ins," after the API reference, for details on how to make your plug-in scriptable.

# Calling Conventions

A BBEdit plug-in's main entry point has the following prototype:

```
OSErr main(BBXTCallbackBlock *callbacks, WindowRef window, long flags,
                AppleEvent *event, AppleEvent *reply);
```

The "callbacks" argument is a pointer to a block of vectors used by the macros in BBXTInterface.h to provide callback services to the plug-in. You should consider this structure read-only and opaque.

The "window" argument is a Mac OS window reference to the window that was in front at the time the plug-ins code was called.

The "flags" argument provides information about BBEdit's current state: it is the logical OR of the following bit masks:

```
#define xfWindowOpen                    0x00000001
#define xfWindowChangeable      0x00000002
#define xfHasSelection          0x00000004
#define xfUseDefaults                   0x00000008
```

The bit masks have the following meanings:

| Bit Name | Meaning |
|---|---|
| `xfWindowOpen` | If set, then the front window is an editing window (or contains a text view, as in the case of a browser window). If clear, then there are no windows open, or the front window does not contain a text view. |
| `xfWindowChangeable` | If set, then the text in the front window is modifiable. If clear, then the front window is read-only (a read-only text file or a browser window). This bit is always clear if xfWindowOpen is clear. |
| `xfHasSelection` | If set, then there is a selection range in the front window. If clear, there is no selection range. This bit is always clear if xfWindowOpen is clear. |
| `xfUseDefaults` | If set, then the plug-in should work without requesting information from the user, either by using stored preferences or by using "factory" defaults. (This bit will be set if and only if the "Use Option Key for Defaults" bit in the 'BBXF' resource is set and the user held down the Option key while choosing the plug-in's name from the Tools menu.) |

The two AppleEvent parameters, "event" and "reply", are used when your plug-in is called from an OSA script. On entry, the "event" parameter points to an Apple Event which contains parameters that your plug-in can interpret for its own needs, and the "reply" parameter points to a reply event that will be returned to the sender of the input event. For more details on writing scriptable plug-ins, see (surprise!) "Writing Scriptable Plug-ins", below.

> **Note:** Your plug-in should allow for the likelihood that both the "event" and "reply" parameters may be `NULL`. This will be the case if your plug-in is run manually by the user (i.e. the old-fashioned way, by choosing it from the Tools menu).

# Programming Interface

*Introduction*

The interface to BBEdit is kept in a structure known as a `BBXTCallbackBlock`. A pointer to this structure is passed on entry to your plug-in's `main`. Although the layout of the callback block is described in `BBXTInterface.h`, we strongly discourage you from using or depending on the layout of this block. `BBXTInterface.h` includes the necessary macro definitions to use the interface structure.

In the interests of convenience and maintenance, all of the documentation on the programming interface may be found in HeaderDoc format in `BBXTInterface.h`. (HeaderDoc is a human-readable, machine-parseable document format. For more, go to <http://developer.apple.com/darwin/projects/headerdoc/>.

The balance of this document deals with topics which are not specifically addressed by the HeaderDoc comments.

# Writing Scriptable Plug-ins

For even greater power and flexibility, you can write plug-ins that are scriptable. This means that users can write OSA scripts which use your plug-in to accomplish an automated task.

There are several considerations to bear in mind when writing a scriptable plug-in:

- You should be familiar with programming Apple Events. If you've ever written an Apple Event-aware application with a scripting dictionary, you're all set. If not, take some time to read the many Develop articles, MacTech Magazine pieces, and chapters of Inside Macintosh that deal with handling Apple Events and writing scripting dictionaries ('aete' resources).

- On entry to your plug-in, the `event` and `reply` parameters will point to AppleEvent structures. The `event` parameter contains incoming Apple Event parameters. The parameters are stored inside the event as defined in the plug-in's scripting dictionary. If you wish, you may store results and (if necessary) error messages in the reply event. Any output data generated by the plug-in should be stored under the `keyAEResult` keyword; an OS result code should be stored as `keyErrorNumber`. If you generate a text error message, it should be stored as `keyErrorString`.

- each 'aete' resource must correspond to a single plug-in. If you are constructing an plug-in file with multiple plug-ins, each plug-in needs to have its own 'aete' resource, whose name needs to match the "MenuName" key of the plug-in's property list.

- Each 'aete' can include only one suite. We recommend that you use 'BBXT' (the BBEdit Plug-in Suite) as the suite code, but there is no requirement that you do so; the suite code may be anything that you choose, as long as it does not conflict with any published suite definition (including suites specified in the scripting dictionaries of BBEdit or other applications).

- Within the suite that you define, the 'aete' resource may contain definitions for as many events as you wish. An example of an plug-in with support for multiple events is the Prefix/Suffix Lines plug-in; although there is only one plug-in, that plug-in responds to four discrete events.

- When your plug-in is scripted (that is, when the `event` and `reply` parameters are not `NULL`), you should avoid any actions that require user interaction, such as dialog boxes or alerts.

- Bear in mind that your plug-in *will* be called with `event` and `reply` parameters of `NULL`. This will occur when the user chooses your plug-in from BBEdit's **Tools** menu.

Within these guidelines, you have a great deal of latitude in specifying the Apple Event interface to your plug-in.