

BBEdit Extensions

BBEdit 2.2 supports the addition of external code modules known as "BBEdit Extensions". This chapter will describe the functionality of the extensions that ship with BBEdition, including:

- 827
- Capitalize Sentences
- Concatenate Files
- Educate Quotes
- Hello World
- Copy Lines Containing
- Cut Lines Containing
- Prefix/Suffix Lines

The section "Writing BBEdition Extensions" will discuss how to write extensions for BBEdition. You can skip this chapter if you aren't a programmer or have no desire to write extensions.

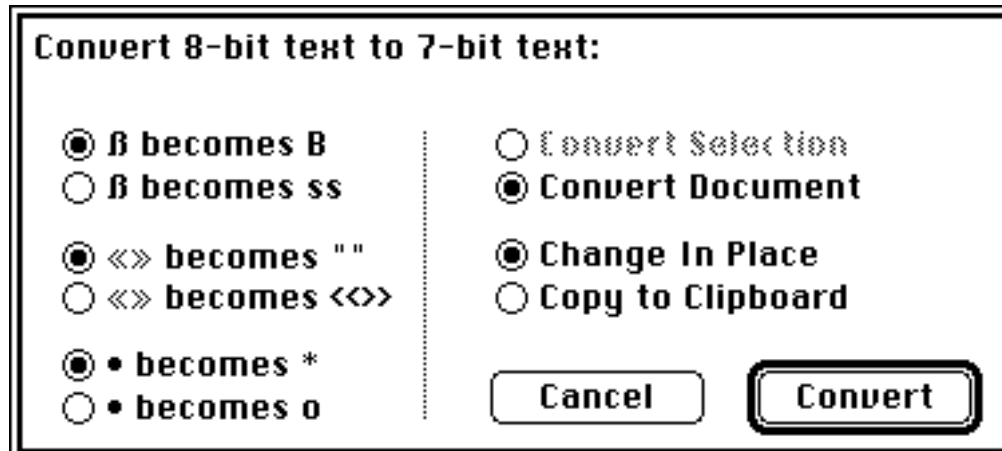
For details on installing the BBEdition Extensions folder, see the "Installing BBEdition" chapter.

General Notes:

- To use an extension, choose its name from the **Extensions** menu. If you do not see this menu in the menu bar, make sure that you have installed the BBEdition Extensions folder correctly.
- Some extensions may require that a document window be open and in front in order to do their work; others may require a selection range in the front document window. If these requirements aren't met, the extensions name in the **Extensions** menu will be dimmed.
- All of these extensions are demos, to show how to write extensions for BBEdition. The source for each one is included, and can be found in the "Sources" folder within the "BBEdition Extensions" folder. Because they are demos, some of the extensions are trivial and don't perform any useful day-to-day function.

827

As its name implies, "827" converts eight-bit special Macintosh characters to reasonable 7-bit ASCII equivalents. This is quite useful for preparing text for posting to an information service that doesn't gracefully handle 8-bit characters. 827 will work on the front document window; when you choose its name from the **Extensions** menu, 827 will present following dialog:



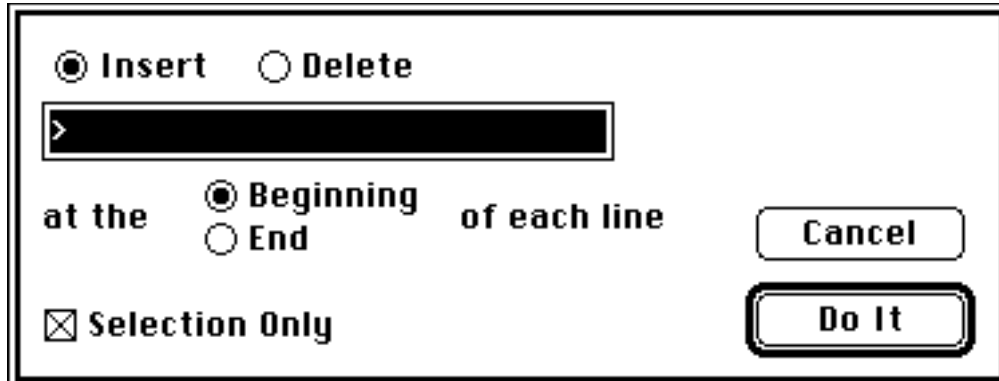
Each pair of radio buttons on the left half of the dialog determines how that particular 8-bit character will be converted. Additionally, curly quotes such as " " and will also be straightened.

The pairs of radio buttons on the right determine what action will be taken. If there is a selection range in the document, "Convert Selection" will be enabled, and if you click on it, only the selection range will be affected. If "Change In Place" on the lower pair of buttons is selected, then the text being converted will be replaced with the converted text; otherwise, the converted text will be placed on the Clipboard for easy pasting into a terminal program, and the text in the document will be unaffected.

827 was written by Jamie McCarthy, based on a prototype "Prepare Usenet Posting" extension by Rich Siegel.

Prefix/Suffix Lines

Prefix/Suffix Lines is useful for adding or removing text from the beginning or end of every line in the document (or selection range).



The "Selection Only" check box will only be enabled if there is a selection range in the document; if its checked, only the text in the selection range will be affected.

The "Insert" radio button will place the text that you enter in the text field at the beginning or end of each line, depending whether "Beginning" or "End" is selected.

The "Delete" radio button will search for the text that you enter in the edit field at the beginning or end of each line, and if its there, will delete it.

This extension is handy for preparing followups to Usenet articles, or for cleaning up downloaded articles.

Writing Extensions

This chapter goes into some technical detail. If you're not a programmer, you can safely skip this section without missing anything.

Introduction

BBEdit version 2.2 has a facility for calling code modules which are not part of the application itself. The main reason for this facility is so that users and third-party programmers can add specific functionality to BBEdition which goes beyond BBEdition'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.

General Guidelines

BBEdit extensions are built as standalone code resources of type 'BBXT'. The capability to build such resources is an integral part of THINK C and THINK Pascal. Users of MPW can also build standalone code resources, but with less ease.

There may be any number of extensions per file, and extensions can use their own resources. Also, BBXT resources can be of any resource ID, since BBEdition manages the extensions in such fashion that resource name or ID conflicts don't happen. Each BBXT resource in a file should have a name as assigned by ResEdit's "Get Info" command; this name will appear under the **Extensions** menu in BBEdition.

Note: If there are BBXT resources from different files with the same name, users may become confused. Neither I nor BBEdition will arbitrate extension names.

When BBEdition starts up, it takes account of the extensions in the "BBEdit Extensions" folder. The BBEdition Extensions folder can reside in the same folder as BBEdition itself. Under System 6, the BBEdition Extensions folder can also reside in the system folder on the startup disk; under System 7, the BBEdition Extensions folder can also reside in the Extensions folder in the system folder on the startup disk.

Files containing extensions must be of type 'BBXT'. The creator can be anything you like, although files with a creator of 'R*ch' will have a standard icon. (You can, of course, create bundle resources and icons to give the files any icons you desire.)

BBEdit extensions should be as friendly as possible. They should take great care to release any memory that they allocate while running, and they should leave no windows on the screen after they return to BBEdition. In general, BBEdition extensions should be considered one-shot text filters: they do their thing, then exit. 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 BBEdition provides.)

You should assume that any callback will move memory. This means that if you keep pointers into any relocatable blocks, pass addresses inside relocatable blocks as function arguments, you should lock the block first. For maximum friendliness, move it high with MoveHHI() first.

Extensions can put up modal dialogs and alerts, provided they're taken down again before the extension exits; they can also call Standard File or any system services necessary, as long as no attempt is made to bring another application to the front.

All other caveats with respect to managing A4 for code resources with globals remain in effect.

Programming Interface

Given all of these constraints, what can extensions do?

The answer is: pretty much any transformation on a window's text that they please.

The interface to BBEEdit is kept in a structure known as an ExternalCallbackBlock. This structure begins with a 16-bit integer which is the version number of the callback block. If the callback block passed to you is higher than one you know about, then there is additional functionality available that you probably don't know about. Conversely, if the version number is less than the one you know about, some functionality that your extension requires may not be available.

The current callback interface version is 2.

Here is the C structure definition for an ExternalCallbackBlock:

```
typedef struct {
    short version;

    //    version 1 callbacks

    pascal Handle      (*GetWindowContents) (WindowPtr w);

    pascal void        (*GetSelection) (long *selStart, long *selEnd,
                                        long *firstChar);

    pascal void        (*SetSelection) (long selStart, long selEnd, long
                                        firstChar);

    pascal void        (*GetDocInfo) (WindowPtr w, Str255 fName, short
                                        *vRefNum, long *dirID);

    pascal long        (*GetModDate) (WindowPtr w);

    pascal Handle      (*Copy) (void);

    pascal Handle      (*Paste) (Handle pasteText);

    //    version 2 callbacks

    /*    Text-Editing stuff */
    pascal long        (*GetLastLine) (void);

    pascal long        (*GetLineNumber) (long selection);

    pascal long        (*GetLineStart) (long selection);

    pascal long        (*GetLineEnd) (long selection);

    pascal long        (*GetLinePos) (long line);

    pascal void        (*Insert) (char *text, long len);

    pascal void        (*Delete) (void);
}
```

```

/*      Getting and Setting window text */
pascal void      (*SetWindowContents) (WindowPtr w, Handle h);

pascal void      (*ContentsChanged) (WindowPtr w);

/*      Reading file text */
pascal Handle    (*GetFileText) (short vRefNum, long dirID,
                                Str255 fName, Boolean *canDispose);

/*      Direct user-interface calls */
pascal Boolean   (*GetFolder) (Str255 prompt, short
                                *vRefNum, long *dirID);

pascal Boolean   (*OpenSeveral) (Boolean sort, short
                                *file_count, StandardFileReply ***files);

pascal DialogPtr (*CenterDialog) (short dialogID);

pascal Boolean   (*StandardFilter) (DialogPtr d, EventRecord
                                *event, short *item);

pascal void      (*FrameDialogItem) (DialogPtr d, short
                                item);

pascal WindowPtr (*NewDocument) (void);

pascal WindowPtr (*OpenDocument) (void);

/*      Utility Routines */
pascal Handle    (*Allocate) (long size, Boolean clear);

pascal long      (*FindPattern) (char *text, long text_len,
                                long text_offset, char *pat, long pat_len,
                                Boolean case_sensitive);

pascal void      (*ReportOSError) (short code);

/*      Preference routines */
pascal void      (*GetPreference) (ResType prefType, short
                                req_len, void *buffer, short *act_len);

pascal void      (*SetPreference) (ResType prefType, short
                                req_len, void *buffer, short *act_len);

} ExternalCallbackBlock;

```

Each field of the callback block is a pointer to a routine. Each routine is called with the Pascal calling convention; in the following descriptions the pascal keyword is omitted for clarity.

```
Handle      (*GetWindowContents) (WindowPtr w);
```

returns a handle to the text in the window pointed to by w. This routine should only be called on windows which have a window kind of userKind.

```
void (*GetSelection)(long *selStart, long *selEnd, long *firstChar);
```

Sets the 32-bit integers pointed to by the arguments to the character offsets of the start of the selection, the end of the selection, and the first visible character in the active editing window.

```
void (*SetSelection)(long selStart, long selEnd, long firstChar);
```

Sets the selection range and first visible character in the active editing window to the values passed. If `firstChar` is -1, the selection range will be centered in the window.

```
void (*GetDocInfo)(WindowPtr w, Str255 *fName, short *vRefNum, short *dirID);
```

Returns information about the window pointed to by `w`. If the window corresponds to a document that doesn't exist on disk, then `fName` will be an empty string, and `vRefNum` and `dirID` will be set to zero. This routine should only be called on windows with a window kind of `userKind`.

```
long (*GetModDate)(WindowPtr w);
```

Returns the modification date (in Macintosh time) of the document whose window is pointed to by `w`. If the document is saved on disk, then the last-modified time of the file is returned; otherwise the time of last edit is returned. This routine should only be called on windows with a window kind of `userKind`.

```
Handle (*Copy)(void);
```

Returns a handle to a copy of the text enclosed by the current selection in the active document. The **caller** is responsible for disposing of this handle when finished with it.

```
Handle (*Paste)(Handle pasteText);
```

Pastes the text in the handle pointed to by `pasteText` into the current selection range of the active document. The **caller** is responsible for disposing of this handle when finished with it.

```
long (*GetLastLine)(void);
```

Returns the number of lines in the active editing document.

```
long (*GetLineNumber)(long selection);
```

Returns the line number of the character offset indicated by `selection`.

```
long (*GetLineStart)(long selection);
```

Returns the character offset of the beginning of the line that `selection` is on.

```
long (*GetLineEnd)(long selection);
```

Returns the character offset of the end of the line that `selection` is on.


```
long (*GetLinePos) (long line);
```

Returns the character offset of the beginning of `line`.

```
void (*Insert) (char *text, long len);
```

Inserts the `len` characters pointed to by `text` in the current selection range of the active editing document.

```
void (*Delete) (void);
```

Deletes the characters enclosed by the selection range in the active editing document.

```
void (*SetWindowContents) (WindowPtr w, Handle h);
```

Replaces the contents of the document designated by `w` with the contents of the handle `h`.

Note: after calling `SetWindowContents`, the handle belongs to the window, and **must not be disposed**. Also, if you modify the contents or size of the handle pointed to by `h` after using it in a `SetWindowContents()` call, be sure to call `ContentsChanged()` for `w`.

```
void (*ContentsChanged) (WindowPtr w);
```

This routine should be called if you directly modify the text returned from a `GetWindowContents()` call.

```
Handle (*GetFileText) (short vRefNum, long dirID, Str255  
fName, Boolean *canDispose);
```

Loads the contents of the designated file's data fork into memory, and returns a handle to those contents. If there was an error (insufficient memory, file system error, etc), `GetFileText()` will return `NIL`.

The `canDispose` argument will be set to `TRUE` if the text was loaded from disk, `FALSE` if the text belongs to an open window. In the event that `canDispose` is `TRUE`, then you should dispose of the text (or use it in a `SetWindowContents()` call). If `canDispose` is `FALSE`, then you **must not dispose the handle**, or else you'll crash BBEdition. Also, you must not modify the contents of the handle if `canDispose` is `FALSE`.

```
Boolean      (*GetFolder)(Str255 prompt, short *vRefNum, long *dirID);
```

Displays a Standard File dialog box for choosing a folder. Returns TRUE if a folder was selected, FALSE if the user clicked the Cancel button. The vRefNum and dirID of the chosen folder are returned in vRefNum, and dirID, respectively.

```
Boolean      (*OpenSeveral)(Boolean sort, short *file_count,  
                          StandardFileReply ***files);
```

Displays a Standard File box for choosing multiple files at once. Returns TRUE if the user chose any files, FALSE if the Cancel button was clicked. If sort is TRUE, then the files returned will be sorted in alphabetical order; otherwise, the files will be returned in the order the user added them to the list.

The number of files chosen will be returned in file_count, and a handle to a list of StandardFileReply records (system 7 style) will be returned in files.

```
DialogPtr    (*CenterDialog)(short dialogID);
```

Loads the dialog box indicated by dialogID and centers it on the screen. The dialog ID should correspond to a dialog which is available in the extension's resource file, and nowhere else. (The resource map chain is configured such that none of your dialog IDs can conflict with BBEdition's.)

```
Boolean      (*StandardFilter)(DialogPtr d, EventRecord *event, short  
                              *item);
```

This standard filter performs some useful standard behavior, such as outlining the default button with a thick border, and handling activates and deactivates for BBEdition's own windows. It is strongly recommended that you pass this pointer as the filterProc argument when calling ModalDialog() or Alert(). If you're writing custom dialog filters in your extension, you should call this routine directly after doing your own preprocessing.

```
void         (*FrameDialogItem)(DialogPtr d, short item);
```

This routine will draw a rectangle around the dialog item specified. If the item is a line, a line will be drawn using true gray.

```
WindowPtr    (*NewDocument)(void);
```

Opens a new untitled document, and returns a pointer to its window. This document becomes the current document. Will return NIL if for some reason the window couldn't be opened.

```
WindowPtr    (*OpenDocument) (void);
```

Puts up BBEdition's standard Open dialog for choosing a file. If the user confirms the dialog and the document is successfully opened, returns a pointer to its window. Will return NIL if the user cancels the dialog or if an error occurred while opening. (If some system error occurs, BBEdition will pose the alert box.)

```
Handle       (*Allocate) (long size, Boolean clear);
```

Allocates and returns a handle of `size` bytes. If the `clear` argument is TRUE, the handle will be zeroed. The handle returned will be a real handle, but may reside in MultiFinder temp memory. As with any handle, you should avoid locking handles returned by `Allocate()` for any length of time, and you should dispose of the handle before returning.

```
long         (*FindPattern) (char *text, long text_len, long text_offset,  
                             char *pat, long pat_len,  
                             Boolean case_sensitive);
```

Searches the text buffer pointed to by `text` for the string of characters pointed to by `pat`. `text_len` is the amount of text to search. `text_offset` is the position relative to the start of the text to start searching. `pat_len` is the length of the string to match. If `case_sensitive` is TRUE, then the case of potential matches will be checked.

`FindPattern()` will return the offset relative to the start of the text that the string was found. If the string was not found, `FindPattern()` will return -1.

```
void         (*ReportOSError) (short code);
```

Displays an alert box with the proper OS error message corresponding to the OS result code given in `code`. This is handy for reporting filesystem errors, out of memory, and things of that sort.

```
void      (*GetPreference) (ResType prefType, short req_len, void
                *buffer, short *act_len);
void      (*SetPreference) (ResType prefType, short req_len, void
                *buffer, short *act_len);
```

The `GetPreference` and `SetPreference` calls are for extensions to use to save and retrieve extension-specific information across runs. The settings are stored in the BBEdition Prefs file as resources.

`GetPreference` will retrieve the preference data stored in the resource of `prefType`, resource ID 128, and copy the contents of that resource into the data pointed to by `buffer`. In all cases, `req_len` represents the maximum number of bytes which will be copied. (**Warning:** the amount of data allocated in `buffer`, be it a static structure or a handle, must be equal to or greater than `req_len`, or else havoc will occur.) The word pointed to by `act_len` will be filled in with the actual number of bytes copied; this is always less than or equal to `req_len`. If `act_len` is negative, the value in `act_len` is an OS error code (usually `resNotFound` if you're calling `GetPreference` with a virgin Preferences file).

`SetPreference` is the complement of `GetPreference`; it writes out the data in `buffer` to a resource of type `resType`, id 128. `req_len` and `act_len` behave as for `GetPreference`.

For examples of how to use the various call backs, look at the sources to the standard extensions in the "Sources" folder, in the "BBEdit Extensions" folder.

Demo Extensions

In addition to 827 and Prefix lines, the following demo extensions are supplied:

Capitalize Sentences

Capitalize Sentences will capitalize the first word in every sentence in the selection range. The entire document will be affected if no selection range exists. This is a very simple extension that demonstrates the use of some of the BBEdition extension facilities.

Concatenate Files

Concatenate Files is a simple extension which demonstrates more of BBEditions extension facilities. This extension poses an "Open Several..." dialog in which you can specify a number of text files. The files you designate will be concatenated and the text of all of them will be placed in a new untitled window (provided that there is enough memory).

Educate Quotes

Educate Quotes is a simple extension which converts straight quotes in your document into "smart" quotes, just as if you had manually gone through the document and re-typed all of the quotation marks with Smart Quotes turned on for the document.

Hello World

Hello World is a trivial extension that creates a new untitled document window with the text "Hello World" in it. It is purely a demo, with no useful function whatsoever.

Copy Lines Containing Cut Lines Containing

These extensions will search through the current document for lines which contain the search string that you enter in the dialog box. Each line found will be placed in the Clipboard. If you use "Cut Lines Containing", each line will also be deleted from the document.