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
internet email `devsupport@barebones.com`
world wide web `http://www.barebones.com/`

v1.5.1 December 18, 2006

Writing Language Modules for BBEdit and TextWrangler

Advisory

Please note that the language module API is subject to change without notice. We are currently working on ways to make it easier for language developers to write language modules, and some of those improvements may require incompatible changes to the API. If that happens, modules written to the old API will have to be revised, and will not be backward compatible; however, we will attempt to minimize the impact on existing language modules, so that you need do only a minimum of work to update your modules.

Introduction

Starting with the release of BBEdit 6.0, it has been possible to write code plug-ins which add syntax coloring and/or function navigation for source files in languages not otherwise supported at the factory. This document provides the technical information necessary for third-party developers to implement such modules, with the intent of improving BBEdit's utility as an editor for source files written in such languages.

Bare Bones also ships TextWrangler, an editor with a different target audience. Both products can use the same language modules. Throughout this document, wherever BBEdit is mentioned, the same functionality is available in TextWrangler.

This document describes how to build a Mach-O Bundled language module using Apple's Xcode IDE. These modules will run in BBEdit 8.0 and later, and TextWrangler 2.0 or later.

Structure of a Language Module

A language module is packaged as a CFBundle, which includes the PowerPC implementation code for the language module, and a plist, which describes all of the module's details.

The bundle signature of a language module file is '`BBLM`'. The sample Xcode project supplied with this documentation is the exact same project file used at Bare Bones to build the Setext language module as shipped with BBEdit 8.0 and TextWrangler 2.0.

The Info.plist

The Info.plist file contains an XML description of the languages supported by the language module, keyword lists, 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 language module specific data. The top-level key "com.barebones.bblminfo" contains an array of one or more dictionaries. Each dictionary should contain the following keys and values:

`BBLMCanGuessLanguage`	A Boolean value which indicates whether or not the language module implements language.
`BBLMColorsSyntax`	A Boolean value which indicates whether this language is syntax-colored, i.e. keywords may be colored (if a keyword list is provided), and the language module will generate and maintain color runs for the source file (see "Syntax Coloring", below).
`BBLMDroppedFilePathStyle`	A String value which allows the module developer to control how BBEdit constructs file paths for insertion into a document when the user drops in a file while holding down the Command key. Must be either "POSIX" or "URL".
`BBLMIsCaseSensitive`	A Boolean value whether this language is case-sensitive (i.e. whether case is relevant for identifiers and keywords). This affects recognition of keywords for syntax coloring.
`BBLMKeywordList`	An Array of String values, one string per keyword.
`BBLMLanguageCode`	A four-character String value used to indicate the language this module should be called for. To override a factory language module, you must use one of the values defined in BBLMTypes.r. If you are providing a language module for a new language, choose a value not in that list.
`BBLMLanguageDisplayName`	A String value with the name of the language this module supports. The string will be used in the BBEdit user interface wherever language names are displayed (the language menu, and the Languages preference panel, for example)
`BBLMMainFunctionName`	Your language module's main entry point (usually main()). Beware of C++ mangling.
`BBLMScansFunctions`	A Boolean value, indicating whether this language has the concept of "functions", i.e. reference points in the text that appear on BBEdit's function popup (and which may be listed in the Current Function display). The term "function"

	is used loosely here; for example, HTML has no concept of subroutines, yet the HTML language module can generate entries in the function popup..
`BBLMUseHTMLFileSearchRules`	A Boolean value used to allow the module developer to control how BBEdit searches for include files chosen from the function popup If set to true, then BBEdit will use the HTML include-file processor (used by the HTML document- and site-update tools) to locate the file. If false, BBEdit will search for the file using the same directory-search strategies as used when using the "Open Selection" command from the File menu. (These strategies are intentionally undefined and subject to change, but suffice it to say that BBEdit makes a good-faith effort to find the file.)
`BBLMSuffixMap`	An Array of Dictionaries describing file suffixes, and their significance to your language. Each Dictionary must contain a `BBLMLanguageSuffix` key, and optionally, *either* a `BBLMIsSourceKind` or `BBLMIsHeaderKind` key. If the language doesn't have the notion of separate extensions for include files, then leave both keys out of the dictionary.
`BBLMLanguageSuffix`	A String value for the filename suffix your language module should be called for. Used only in `BBLMSuffixMap` dictionaries.
`BBLMIsSourceKind`	A Boolean value if the filename suffix is unique to implementation files. Used only in `BBLMSuffixMap` dictionaries.
`BBLMIsHeaderKind`	A Boolean value if the filename suffix is unique to header files. Used only in `BBLMSuffixMap` dictionaries.

Language Module Construction

Main Entry Point

A BBEdit language module consists of a single entry point, traditionally named: `main`. Your main entry point can be named whatever you like. Just be certain that the argument list is identical to the one described below.

When BBEdit attempts to load a language module, it uses CFBundle to locate the main entry point (as decribed in the `BBLMMainEntryPoint` entry in the plist). Beware of C++ name mangling if you are compiling your module with a C++ compiler.

The main entry point is defined as follows:

```
OSErr main(BBLMParamBlock &params,
               const BBLMCallbackBlock &bblm_callbacks,
               const BBXTCallbackBlock &bbxt_callbacks);
```

The parameters to `main()` are as follows:

`params` a `BBLMParamBlock` contains information about the text being processed, what the language module is expected to do on this invocation, and where the results (if any) are to be returned.

`bblm_callbacks`

these language-module callbacks provide application services to support various language module functions. Although the structure of the callback block is provided in `BBLMInterface.h`, you should treat this structure as opaque, and neither read nor write it. Inline functions are provided in `BBLMInterface.h` so that you can use the callbacks.

`bbxt_callbacks`

this callback block provides access to the standard BBEdit callback set (the same as provided to BBEdit plug-ins).

Note: The text passed in to your language module should be considered read-only, so do not use any BBXT callbacks that modify the text. In addition, language modules are not able to interact with the user, or with other applications, so you should not use any BBXT callbacks that do so.

Module Result Codes

The main entry point of a language module returns a Mac OS result code. Typically, if all goes will this will be `noErr`. If a language module callback returns a result code other than noErr, your language module's `main` should return that result code (provided that the callback is defined to return an `OSErr` result).

Parameters

The exchange of data between a language module and BBEdit is accomplished by means of a parameter block. This is a C++ union (or variant record, if you prefer the Pascal nomenclature) which contains input parameters for various operations.

This section provides an overview of all of the members of the parameter block; the sections that follow provide detailed information on each operation and the semantics of the parameters.

Inputs

The following parameters are provided whenever the language module is called. Those which are meaningless for certain operations are usually `NIL` or zero.

`fSignature`	always equal to `kBBLMParamBlockSignature`; if it is not, the parameter block is invalid (or your language module is incompatible) and your module should return `paramErr`.
`fVersion`	version of the parameter block. Ideally it should be equal to the value of `kBBLMParamBlockVersion` as of when your module was compiled. If it is less, you should proceed with care (and you may be conservative and just return `paramErr`), because some parameters may not be valid. If `fVersion` is greater than `kBBLMParamBlockVersion`, then you can proceed safely.
`fMessage`	message code describing the operation being requested by BBEdit. This will be one of the messages listed in the `BBLMMessage` enumeration. If it is less than zero, greater than `kBBLMLastMessage`, or not a message that your module understands, your module should return `paramErr`.
`fLanguage`	If your module is capable of processing multiple languages, you should examine `fLanguage` in order to determine how to proceed. This value is determined by mapping the document's suffix according to the "Languages" preferences panel, and will always be one of the languages specified in your module's plist.
`fText`	points to the text that your module is being asked to process. In the case of a function scan or run calculation, this will point to the beginning of the source file's text. In the case of a keyword lookup, `fText` points to the language keyword. For any operation which doesn't involve any text processing (such as color or run code mapping), `fText` will be `NIL`.
`fTextLength`	if `fText` is not `NIL`, indicates the number of characters pointed to by `fText`. Note that the number of characters is **not** the number of bytes; if `fTextIsUnicode` is `TRUE` (which should always be the case), then `fText` points to an array of Unicode characters, each of which occupies two bytes in memory.
`fTextIsUnicode`	beginning with BBEdit 8 (and TextWrangler 2.0), this parameter is always true, indicating that `fText` points to an array of Unicode characters (each of which occupies two bytes in memory). Additionally, `fTextScript` may be something other than the Script Manager constant `smRoman` (which would indicate that the text is written in a non-Roman writing system).

`fTextScript` — if `fText` is not `NIL`, use this field to determine the writing system ("script", in Mac OS API parlance) in which the text is written. If it is something other than `smRoman`, the text is written in a non-Roman system (for example, Japanese or Chinese), and characters above the standard ASCII range of 0-127 may require special handling.

`fTextGapLocation`
`fTextGapLength` — In order to improve performance, BBEdit's text engine uses a "gap". The gap is a range of characters in the middle of the text which does not appear on the screen, and which is occasionally moved or resized as necessary. So, consider the following text:

```
abcdef····ghij
```

Each bullet (•) indicates a character which is in the gap. On the screen, the text would display:

```
abcdefghij
```

To maintain editing performance when performing syntax-coloring operations, BBEdit leaves the gap in place when calling language modules. (Otherwise, typing performance would be very poor indeed!) However, if you don't know where the gap is, your language module's code cannot account for the gap, and might end up processing whatever junk characters lie inside the gap.

Thus, for certain calls to your language module, `fTextGapLocation` and `fTextGapLength` describe the location and length of the gap. `fTextGapLocation` is the offset, relative to `fText`, of the first character in the gap, and `fTextGapLength` is the number of characters in the gap. In the above example, `fTextGapLocation` is 6, and `fTextGapLength` is 4.

If `fTextGapLength` is zero, then there is no gap in the text (the characters are all contiguous in memory).

Note: `fTextGapLength` is the **only** reliable way to tell if the text contains a gap; it's possible that the gap might be at the very beginning of the text, in which case `fTextGapLocation` will be zero.

If you're iterating through a range of characters, accounting for the gap is easy. Consider the following code (in order to simplify, we're assuming that `fTextIsUnicode` is false):

```
void ProcessTextRange(BBLMParamBlock &pb, UInt32 start,
UInt32 end)
{
   UniChar     *p;         //    points to next character
   UInt32      pos;        //    current character offset
   bool        past_gap;   //    have we passed the
                           //    gap yet?

   past_gap = false;
   p = pb.fText;

   for (pos = start; pos < end; pos++, p++)
   {
```

```
                        //    adjust for the gap if we're not
                        //    already past it

                        if ((! past_gap) &&
                              (pos >= pb.fTextGapLocation))
                        {
                              p += pb.fTextGapLength;
                              past_gap = true;
                        }

                        //    do something to the character
                        whatever(*p);
                  }
            }
```

The gap does complicate your code slightly; to make life easier, we've provided a `BBLMTextIterator` class, which makes it much easier to write gap-aware and multi-byte-aware code. The sample plug-ins supplied with this SDK make use of `BBLMTextIterator` ; please refer to their source code to see how to use it.

`fFcnParams` This member of the union is only meaningful when `fMessage` is `kBBLMScanForFunctionMessage`. It contains information that your language module needs in order to return function information to BBEdit. See "Scanning for Functions," below, for details on the members of `fFcnParams` and how to use them.

`fAdjustRangeParams`
`fCalcRunParams`
`fAdjustEndParams` These members of the union are only meaningful when fMessage is `kBBLMAdjustRangeMessage`, `kBBLMCalculateRunsMessage` or `kBBLMAdjustEndMessage`, respectively. They contain information that your language module needs in order to generate or update syntax coloring runs and return the information to BBEdit. See "Generating Syntax Runs" below for details on the members of `fAdjustRangeParams`, `fCalcRunParams` and `fAdjustEndParams` and how to use them.

`fMapRunParams` This member of the union is only meaningful when `fMessage` is `kBBLMMapRunKindToColorCodeMessage`. It contains information that your language module needs to map a run kind to a color code. See "Advanced Topics," below, for details on the members of `fMapRunParams` and how to use them.

`fMapColorParams` This member of the union is only meaningful when `fMessage` is `kBBLMMapColorCodeToColorMessage`. It contains information that your language module needs to map a color code to an actual RGB color. See "Advanced Topics," below, for details on the members of `fMapCodeParams` and how to use them.

`fCategoryParams` This member of the union is only meaningful when `fMessage` is `kBBLMSetCategoriesMessage`. It contains information that your language module needs in order to customize BBEdit's treatment of word characters when drawing files colored in a language handled by your module. See "Advanced Topics," below, for details on the members of `fCategoryParams` and how to use them.

`fMatchKeywordParams`

This member of the union is only meaningful when `fMessage` is `kBBLMMatchKeywordMessage`. It contains information that your language module needs in order to match keywords that are not specified in the static keyword table for a language handled by your module. See "Advanced Topics," below, for details on the members of `fMatchKeywordParams` and how to use them.

`fEscapeCharParams`

This member of the union is only meaningful when fMessage is `kBBLMEscapeStringMessage`. It contains information that your language module needs in order to generate customized character escapes for display in the "ASCII Table" floating window. See "Advanced Topics," below, for details on the members of `fEscapeCharParams` and how to use them.

Messages

This section provides an overview of the messages that BBEdit may send to your language module.

`kBBLMInitMessage` BBEdit will call your language module with this message when it is loading your language module during application startup. Your module will receive only one call of `kBBLMInitMessage` per execution of the application. You can use the opportunity to initialize any module globals, or perform any other sort of preparatory housekeeping.

`kBBLMDisposeMessage`

BBEdit will call your module with this message when it is unloading your language module during application shutdown. Your module will receive only one call of `kBBLMDisposeMessage` per execution of the application. You can use the opportunity to undo any work done during module initialization (see `kBBLMInitMessage`), or perform any sort of final housekeeping.

`kBBLMScanForFunctionMessage`

BBEdit will call your module with this message whenever it needs a list of all of the functions in the file. Your module should generate a list of functions using the techniques described in "Scanning for Functions," below. Note that your module will only be called with this message if it supports function scanning for its language (i.e. `BBLMScansFunctions` is true in the Info.plist).

`kBBLMCalculateRunsMessage`

BBEdit will call your module with this message whenever it needs a complete list of all syntax-coloring runs for a particular range of text. This usually occurs when loading a file for the first time, changing the language mappings in the Preferences, or when a file's contents are changed wholesale (as in a "Replace All" or other large-scale text transformation).

`kBBLMAdjustRangeMessage`
`kBBLMCalculateRunsMessage`
`kBBLMAdjustEndMessage`

BBEdit will call your module with these messages whenever some incremental change has been made to the text, and it needs your language module to recalculate the syntax coloring runs to account for the change. This usually occurs when the user types, or performs a Cut or Paste operation. In order to preserve typing performance, your module should recompute the smallest possible range of text consistent with correct syntax coloring. This can be tricky; the "Syntax Coloring" section, below, describes some techniques, and the example language module(s) may provide a good starting point.

`kBBLMMapRunKindToColorCodeMessage`

BBEdit will call your module with this message when it encounters a syntax run whose run kind is in the range of user-defined syntax run kinds. See "Advanced Topics," below, for more information.

`kBBLMMapColorCodeToColorMessage`

BBEdit will call your module with this message when it encounters a color code in the range of user-defined color codes. See "Advanced Topics," below, for more information.

`kBBLMSetCategoriesMessage`

BBEdit will call your module with this message to set up a custom word-break table, in order to faciliate correct recognition of nonstandard word boundaries when drawing text. You generally only need to support this message if the language that you're coloring contains keywords whose component characters are ordinarily considered word breaks. See "Advanced Topics," below, for more information.

`kBBLMMatchKeywordMessage`

BBEdit will call your module with this message in order to match a token which might be a language keyword but which does not appear in the static keyword table provided by your module. See "Advanced Topics," below, for more information.

`kBBLMEscapeStringMessage`

BBEdit will call your module with this message when it wants to generate an escape sequence for an otherwise unprintable ASCII character. This is currently used only by the ASCII Table floating window, and support for this message is strictly optional.

A Note About Messages and Result Codes

If your language module is called to perform a function that it does not support, e.g. a syntax-coloring message passed to a module that only supports function scanning, or *vice versa*, this may indicate a misconfiguration of your module's Info.plist. However, if this does occur, your module should return `paramErr` to indicate that the module was called with unexpected or inconsistent parameters.

In some situations, your language module may receive a call to perform an action that is not strictly required for a language that it's processing. If this occurs, your language module should perform no action and return `noErr` as its result code. This is particularly important when processing a `kBBLMAdjustRangeMessage` in a module that supports syntax coloring: if you return something other than `noErr`, syntax coloring information will be out of sync with the text.

Scanning for Functions

In this section, we use the term "function" to refer to anything that may appear on BBEdit's function popup. In source files for compiled languages, a "function" may refer not only to a subroutine, but to a typedef, function prototype, a special compiler directive, or anything you think the user might find useful to have on the function popup. Since non-compiled languages (e.g. markup languages like SGML or TeX, or structure languages like Setext) don't have any concept of functions or subroutines, the term "function" refers to any useful reference point (such as a section break in TeX or Setext).

If your language module supports function scanning, it will be called with a `kBBLMScanForFunctions` message. In the paramter block, `fText` will point to the text to be scanned (the beginning of the source file), `fTextLength` will indicate the number of Unicode characters pointed to by `fText`.

Input Parameters

When scanning for functions, the `fFcnParams` member of the input `BBLMParamBlock` contains additional parameters that you'll use to generate the function list:

`fTokenBuffer`	an opaque reference to a data structure which holds the names of items on the function popup menu. `fTokenBuffer` is passed as the first argument to the callback function `bblmAddTokenToBuffer`.
`fFcnList`	an opaque reference to a data structure which contains a list of `BBLMProcInfo` structures (see below), each one corresponding to an item on the function popup menu. `fFcnList` is passed as the first argument to the callback functions `bblmAddFunctionToList`, `bblmGetFunctionEntry`, and `bblmUpdateFunctionEntry`.
`fOptionFlags`	a collection of flag bits which controls optional behaviors. At present, the only flag used is `kBBLMShowPrototypes`. If this flag is set (i.e. (`fOptionFlags & kBBLMShowPrototypes`) is nonzero), then the "Show Prototypes" preference is turned on, and you should generate appropriate function entries for function prototypes, if your language supports them.

Logic Overview

When scanning a source file for functions, your module will follow this general scheme:

```
while (there are still characters left)
{
      find a function's start and end offsets;

      use bblmAddTokenToBuffer or bblmAddCFStringTokenToBuffer to
               add the function's identifier to the token buffer;

      fill in a BBLMProcEntry structure with information about
               the function;

      use bblmAddFunctionToList to add the function information
               to the function list;
}
```

The complexity of finding a function's start and end offsets varies depending on the particular language. You will of course need to take care to process strings and character escape sequences correctly, and there may be other considerations involved in correctly identifying structural elements.

In some languages, you may need to go back and "fix up" a function entry that you've previously added to the function list. This is most likely to happen in languages that allow nesting of certain structures. (For example, PHP allows you to nest functions inside of classes, *ad infinitum*.) In other languages, you can't know where the end of one function is until you've found the start of the next one (Setext is an example of such a language).

To handle these situations, the callback interface provides routines to obtain a copy of a function entry from the function list, and to update a function entry that's already in the list. With these services, you can implement function scanning as follows, if you need to:

```
UInt32         fcnIndex = 0;

while (there are still characters left)
{
      find a function's start;

      if (fcnIndex > 0)
      {
               use bblmGetFunctionEntry to get the entry for for
               the (fcnIndex-1)-th function;

               set the entry's fFunctionEnd to the current
               function start;

               use bblmUpdateFunctionEntry to update the entry in
               the list;
      }

      use bblmAddTokenToBuffer or bblmAddCFStringTokenToBuffer to
               add the function's identifier to the token buffer;

      fill in a BBLMProcEntry structure with information about
               the function;
```

```
        use bblmAddFunctionToList to add the function information
                to the function list;

        fcnIndex++;
    }
```

The supplied source code for the Setext language module shows how to do this in a real situation.

The BBLMProcInfo structure

```
typedef         struct
{
        UInt32   fFunctionStart;//    char offset in file of first
                                //    character of function
        UInt32   fFunctionEnd;  //    char offset of last character of
                                //    function

        UInt32   fSelStart;     //    first character to select
                                //    when choosing function
        UInt32   fSelEnd;       //    last character to select when
                                //    choosing function

        UInt32   fFirstChar;    //    first character to make visible
                                //    when choosing function

        UInt32   fIndentLevel;  //    indentation level of token
        UInt32   fKind;         //    token kind
        UInt32   fFlags;        //    token flags
        UInt32   fNameStart;    //    char offset in token buffer of token
                                //    name
        UInt32   fNameLength;   //    length of token name
} BBLMProcInfo;
```

The `BBLMProcInfo` structure contains information about a single entry in the function menu. The members of this structure are as follows (character offsets are zero-based and relative to the beginning of the file):

`fFunctionStart` the offset of the first character in the function

`fFunctionEnd` the offset of the last character in the function

`fSelStart` the offset of the first character to select when the function is chosen from the function popup

`fSelEnd` the offset of the last character to select when the function is chosen from the function popup

`fFirstChar` the offset of the first character to make visible when the function is chosen from the function popup. This is generally the same as `fFunctionStart`; however, if the function is immediately preceded by a comment, you may wish to have `fFirstChar` point to the start of the comment, so that both the comment and the function header are visible when the user selects the function.

`fIndentLevel` indicates the nesting depth of the function; top-level functions have an `fIndentLevel` of zero. BBEdit will indent the function name on the menu by an amount corresponding to the value of `fIndentLevel`.

`fKind` indicates the type of function, based on the following enumeration:

```
{
      kBBLMFunctionMark,
      kBBLMTypedef,
      kBBLMPragmaMark,
      kBBLMInclude,
      kBBLMSysInclude
}
```

For most subroutines and section elements, you'll use `kBBLMFunctionMark` for most items, and `kBBLMTypedef` for structured type declarations in compiled languages. If your language supports a `#pragma mark`– style convention for embedding markers in the form of special comments or preprocessor directives, then use `kBBLMPragmaMark` to indicate such an item.

`kBBLMInclude` and `kBBLMSysInclude` indicate the presence of an include-file specification. If your language supports includes, then you should place the name of the include file in the token buffer, and set the function's kind to `kBBLMInclude`, or, if the language supports "system includes" akin to C's "`#include <foo.h>`" syntax, use `kBBLMSysInclude`. Note that BBEdit *does* differentiate between `kBBLMInclude` and `kBBLMSysInclude`; when you choose the name of an include file from the function popup, BBEdit will search the system tree (as specified in the "File Search" preferences) for the file, to the exclusion of any other directories.

Callbacks

The following callbacks are provided for building the function list:

```
OSErr bblmAddTokenToBuffer(const BBLMCallbackBlock *callbacks,
                           UInt32 tokenBuffer,
                           void *id,
                           UInt32 length,
                           bool isUnicode,
                           UInt32 *offset);

OSErr bblmAddCFStringTokenToBuffer
                           (const BBLMCallbackBlock *callbacks,
                           UInt32 tokenBuffer,
                           CFStringRef id,
                           UInt32 *offset);
```

The token buffer is a data structure which contains the text of all items that appear on the function menu. `BBLMParamBlock.fFcnParams.fTokenBuffer` contains an opaque reference to the token buffer in which BBEdit stores function names; you should pass this value as the `tokenBuffer` argument. For `bblmAddTokenToBuffer, id` is a pointer to the token string, i.e. the identifier. `length` is the length, in characters, of the identifier. `isUnicode` indicates whether the token pointed to by `id` is stored as Unicode text; this is usually the case when the document being scanned is stored as multi-byte text. (You can eliminate these considerations by creating a CFStringRef and passing it to `bblmAddCFStringTokenToBuffer`.) If `bblmAddTokenToBuffer` returns `noErr`, then the value pointed to by `offset` will be filled in with an offset into the token buffer. This value should be used as the `fNameOffset` field when constructing a `BBLMProcInfoRecord`.

```
OSErr bblmAddFunctionToList(const BBLMCallbackBlock *callbacks,
                            UInt32 procList,
                            BBLMProcInfo &info,
                            UInt32 *index);
```

This function, along with `bblmAddTokenToBuffer`, is the backbone of your function scanner; given a `BBLMProcInfo` structure, `bblmAddFunctionToList` adds it to the list of items to be displayed in BBEdit's function popup. The `procList` argument is an opaque reference to BBEdit's internal function list; you should pass `BBLMParamBlock.fFcnParams.fFcnList` as this argument. `info` is your `BBLMProcInfo` structure with all fields filled in. If this callback returns with a result of noErr, the value pointed to by `index` is the position in the array of this function record. If necessary, you can pass this value to `bblmGetFunctionEntry` and `bblmUpdateFunctionEntry`.

```
OSErr bblmGetFunctionEntry(const BBLMCallbackBlock *callbacks,
                           UInt32 procList,
                           UInt32 index,
                           BBLMProcInfo &info);
```

Use `bblmGetFunctionEntry` to retrieve a particular item from the function list. As with `bblmAddFunctionToList`, the `procList` argument is an opaque reference to BBEdit's internal function list; you should pass `BBLMParamBlock.fFcnParams.fFcnList` as this argument. `index` is the (zero-based) index of the function entry you wish to retrieve; it should be between zero and the number of items in the function list. If `bblmGetFunctionEntry` returns `noErr`, the BBLMProcInfo record refernced by `info` argument will be filled in with the relevant information.

```
OSErr bblmUpdateFunctionEntry(const BBLMCallbackBlock *callbacks,
                              UInt32 procList,
                              UInt32 index,
                              BBLMProcInfo &info);
```

Use `bblmUpdateFunctionEntry` to modify an entry in the list. The argument semantics are similar to `bblmGetFunctionEntry`, with the important exception that the info argument should contain a properly-specified `BBLMProcInfo` structure, and will be stored in the function list at the specified `index`. Note that `bblmUpdateFunctionEntry` **cannot** be used to add a new function to the list; you should only call to modify a function entry that you have added with `bblmAddFunctionToList`, or to modify a function entry that you have retrieved with `bblmGetFunctionEntry`.

Syntax Coloring

Syntax Coloring refers to the automatic highlighting of certain textual elements (e.g., comments, keywords, etc.) by coloring them specially. This feature is typically applied to source code written in some programming language, but that is by no means its only application.

At its core, BBEdit has no understanding of syntax, short of assumptions it makes about how words and lines are formed. It does however allow a language module, using its understanding of syntax, to divide a document up into a series of *syntax runs*. Each syntax run is designated as being not of a particular color, but of a particular flavor, or *kind*. BBEdit, in turn, maps each run kind to a particular color based on user preferences.

Generating Syntax Runs

Each syntax run is specified as follows:

```
typedef struct
{
      DescType language;
      short    kind;
      long     startPos;
      long     length;
} BBLMRunRec;
```

A run's `language` should be the same as the language code of the language module that generates it. Some of BBEdit's built-in language modules currently handle multiple languages in a document, but there is currently no way for external language modules to cooperate in order to achieve similar effects.

A run's `kind` can be anything, really, although there are several constants that are predefined for BBEdit's built-in language modules.

```
kBBLMRunIsCode
kBBLMRunIsPreprocessor
kBBLMRunIsPostPreprocessor
kBBLMRunIsBlockComment
kBBLMRunIsLineComment
kBBLMRunIsSingleString
kBBLMRunIsDoubleString
```

What's important about a run's `kind` is that each kind maps to a particular color code. The color code in turn determines what color to apply to the run. The predefined run kinds map to color codes as follows:

Run Kind	Color Code
`kBBLMRunIsCode`	`kBBLMTextColor`
`kBBLMRunIsPreprocessor`	`kBBLMKeywordColor`
`kBBLMRunIsPostPreprocessor`	`kBBLMTextColorNoKeywords`
`kBBLMRunIsBlockComment`	`kBBLMCommentColor`
`kBBLMRunIsLineComment`	`kBBLMCommentColor`
`kBBLMRunIsSingleString`	`kBBLMStringColor`
`kBBLMRunIsDoubleString`	`kBBLMStringColor`

Note that more than one run kind can map to the same color code. You can define your own run kinds and have them map to any of the predefined color codes, or you can define your own color codes and have your run kinds map to those instead. See "Advanced Topics," below, for more details about color codes.

A run's `startPos` is the character offset of the first character in the run. The offset of the first character in a document is zero and the offset of the last character is the length of the document minus one. A run's `length` is the number of characters in the run (be aware that a character is not necessarily the same as a byte).

A series of runs should cover the entire document with no gaps or overlapping. More precisely, for any two adjacent runs the difference between their `startPos` offsets should be **exactly** the length of first one, which should be the one with the lower `startPos`. The offset of the very first run for the document should be zero and the offset of the very last one plus its length should equal the length of the entire document.

When a series of runs for a document is being created for the very first time or being recreated from scratch, your language module will receive a `kBBLMCalculateRunsMessage`. Such messages are accompanied by the following parameters:

```
typedef struct
{
      SInt32   fStartOffset;
      DescType fLanguage;
} bblmCalcRunParams;
```

`fStartOffset` will always be zero when creating a run list from scratch, and `fLanguage` will always be the language module's language code as well. Your language module simply scans the document from beginning to end, informing BBEdit of each run via the `bblmAddRun` callback:

```
bool bblmAddRun(const BBLMCallbackBlock *callbacks,
                DescType language,
                BBLMRunCode kind,
                SInt32 startPos,
                SInt32 length,
                bool dontMerge = false);
```

`callbacks` is a pointer to the `BBLMCallbackBlock` passed to the language module's entry point. `language, kind, startPos` and `length` are the values of the four `BBLMRunRec` fields for this run. `dontMerge` defaults to false and is only ever set to true in special situations that will be discussed in detail in the next section, "Updating Syntax Runs".

`bblmAddRun` returns `false` if the call was unsuccessful (usually due to a memory allocation error) or if BBEdit has otherwise determined that you should stop scanning (for reasons that will become apparent in the next section), in which case you should clean up and return. Otherwise, the run was successfully added to the run list and you should continue scanning (unless you've reached the end of the document, of course).

Updating Syntax Runs

A run list is created from scratch when a document is first opened and after editing operations that have wide-ranging effects (such as a Replace All). Smaller editing operations, such as a minor Cut or Paste or even typing a single character, may invalidate most or all of the existing runs, but more often only a small number of runs are affected. The run list could be easily rehabilitated by tossing it out and generating a new one from scratch after each such operation.

However, except for fairly small files, this would result in major overhead and significant delays between characters while typing.

BBEdit identifies the smallest sequence of runs out of the entire list that are directly affected by the operation and calls you with a `kBBLMCalculateRunsMessage` whose `fStartOffset` parameter is set to the `startPos` of the first affected run (and `fLanguage` is set to its `language).` As before, your language module simply scans the document, this time starting from `fStartOffset,` informing BBEdit of each run via the `bblmAddRun` callback. Scanning should continue until either the end of the document is reached or `bblmAddRun` returns `false` .

When updating syntax runs, it is not necessarily an error for `bblmAddRun` to return `false` (although it may be); it is also used as an indication that the run you have just added matches a run already in the list and that it is unnecessary to scan any further. In fact, you *must* return from the language module at this point. Before it returns `false` , `bblmAddRun` will have pieced together the previous run list with the runs you have added, and any additional calls to `bblmAddRun` will append their runs to the very end of the run list – probably not the desired result.

Identifying Which Runs to Update

Before BBEdit calls you with a `kBBLMCalculateRunsMessage` to update a run list, it will make a call with a `kBBLMAdjustRangeMessage` to give the language module an opportunity to adjust the range of syntax runs affected, if necessary.

A `kBBLMAdjustRangeMessage` call will be accompanied by the following parameters:

```
typedef struct
{
      SInt32      fStartIndex;
      SInt32      fEndIndex;
      SInt32      fOrigStartIndex;
      BBLMRunRec  fOrigStartRun;
} bblmAdjustRangeParams;
```

`fStartIndex` and `fEndIndex` are values that are the beginning and end, respectively, of the range of syntax runs BBEdit has determined to be affected by an editing operation. You may alter these values, but neither should be set to less than zero or greater than the total number of syntax runs. You can determine the total number of syntax runs with the `bblmRunCount` callback:

```
SInt32 bblmRunCount(const BBLMCallbackBlock *callbacks);
```

You should also ensure that `fStartIndex <= fEndIndex`.

`fOrigStartIndex` and `fOrigStartRun` are included primarily for the benefit of the TeX language module and it is unlikely that anyone else would have much interest in them. Nevertheless, they are described in a bit more detail in "Advanced Topics," below.

Your language module's `kBBLMAdjustRangeMessage` handler may examine the syntax runs in the vicinity of `fStartIndex` and/or `fEndIndex` to decide whether to alter them. You may get the values contained in an individual syntax run via the `bblmGetRun` callback:

```
bool bblmGetRun(const BBLMCallbackBlock *callbacks,
                SInt32 index,
                DescType& language,
                BBLMRunCode& kind,
```

```
                    SInt32& charPos,
                    SInt32& length);
```

`index` is the index of the desired syntax run. `language`, `kind`, `charPos` and `length` are references to variables where the values of the four `BBLMRunRec` fields for this run are stored.

Note that the `kBBLMAdjustRangeMessage` handler may choose to do nothing and return noErr right away. However, it should otherwise only make its decisions based on its examination of the run list. It should not need to examine the text itself at this stage. In fact it cannot, even if it wants to – the `BBLMParamBlock` field `fText` is `nil` for all `kBBLMAdjustRangeMessage` calls.

An Example

It is difficult to describe why a module might want to adjust the beginning and end of the range. An example from one of the built-in language modules is perhaps the best way to illustrate this.

In the C and C++ languages there are preprocessor directives, one of the most commonly used being the `#include` directive, e.g.:

```
#include "foo.h"
```

BBEdit's built-in C/C++ language module wants everything following the initial directive name in a directive (except for the `#define` directive) to be colored as plain text. It therefore creates a run of kind `kBBLMRunIsPreprocessor` for the initial `#include` in the above example and a run of kind `kBBLMRunIsPostPreprocessor` for the entire remainder of the directive.

When an edit occurs within the `kBBLMRunIsPostPreprocessor` run, BBEdit determines that it is the only one affected and makes the `kBBLMAdjustRangeMessage` call with the `fStartIndex` parameter set to the offset of the space character following the `#include`. When the handler for the `kBBLMCalculateRunsMessage` begins scanning at this offset, it very quickly encounters a double-quote character and decides to start a `kBBLMRunIsDoubleString` run – which is not the desired result.

To avoid this, the built-in module has a handler for the `kBBLMAdjustRangeMessage` that checks to see if the first run affected by the edit happens to be a `kBBLMRunIsPostPreprocessor` run. If so, it tells BBEdit to back up and include the preceding `kBBLMRunIsPreprocessor` run as well. When the `kBBLMCalculateRunsMessage` is subsequently received, it will have its `fStartOffset` parameter set to the offset of the '#' character that begins the directive. Starting the scan from this point will correctly generate a `kBBLMRunIsPreprocessor` run followed by a `kBBLMRunIsPostPreprocessor` run.

Note that it is conceivable that you could simply assume that the `kBBLMRunIsPostPreprocessor` run will remain a `kBBLMRunIsPostPreprocessor` run and begin scanning with that in mind. A `kBBLMRunIsPostPreprocessor` run will only ever be preceded by a `kBBLMRunIsPreprocessor` run, after all. This case is a comparatively simple one, though. There are other situations in other languages where more contextual information from the preceding text is needed to determine how to correctly proceed with the scan.

BBEdit's built-in language modules have adopted an approach in which they use the same scanning code whether they are generating a complete run list from scratch or updating one. Enabling that code to begin scanning at any arbitrary point in the text would introduce considerable complexity and/or overhead. Instead, BBEdit uses the `kBBLMAdjustRangeMessage` to back up in the text to a point where a scan may proceed forward unambiguously.

Another Example

The situations where you'd want to extend the end of the range are probably rare and, again, somewhat difficult to describe. Another example from one of the built-in language modules is in order.

In the Perl language there is the notion of HERE-DOC text, a text literal that does not appear immediately where it is used. Rather, a tag is placed at the usage point and the text appears on one or more lines following the line containing the tag. The end of the text is signaled by repeating the tag on a line by itself (for this discussion, we'll ignore the possibility of more than one HERE-DOC tag appearing on a single line, thank you very much).

When an edit occurs entirely within a HERE-DOC tag (the one preceding the text, not the one terminating it), the result will often remain a valid HERE-DOC tag. When the scanner responds to a `kBBLMCalculateRunsMessage`, a first call to `bblmAddRun` will probably return false and nothing will really change much. However, because the value of the tag has changed, the terminating tag will no longer match. The net result is that HERE-DOC text is extended until a matching occurrence of the tag is found or, more likely, the end of the text is reached.

The built-in Perl language module therefore has a `kBBLMAdjustRangeMessage` handler that checks to see if a HERE-DOC tag is being rescanned and, if so, advances `fEndIndex` forward to include the HERE-DOC text as well.

Delaying Run List Synchronization

There may be occasions where you may not want the old run list to be checked for a match because to find a match and merge the old and new run lists together would lead to erroneous results. A prime example of this again involves HERE-DOC in Perl.

This time, imagine a line of text where a HERE-DOC tag does not exist and an editing operation that creates one there. Because no HERE-DOC tag was previously present, the `kBBLMAdjustRangeMessage` handler will not find any reason to advance the `fEndIndex` beyond the index of the run being edited. Usually, the run in question will be a `kBBLMRunIsCode` run extending to the end of the line and beyond, and the scanner will naturally cut off the run at the end of the line and gather up text on subsequent lines as a HERE-DOC run. However, if there is a non-code run such as a comment or string literal after the tag, `bblmAddRun` will want to stop the scan at that point.

What the Perl scanner does when it sees a HERE-DOC tag is to set a flag indicating that a HERE-DOC run is pending. When it reaches the end of the line and finds the flag set, it closes off the last run being scanned and begins scanning text into a HERE-DOC run. Also, while that flag is set, calls to `bblmAddRun` are made with the `dontMerge` parameter set to **true**. This tells the callback to ignore any potential match with the old run list and remain prepared to accept more runs. When the HERE-DOC run is added, the flag is cleared and the `dontMerge` parameter is set to **false** and synchronization is again possible.

Adjusting the Endpoint of Redrawing

After BBEdit calls you with a `kBBLMCalculateRunsMessage` to update a run list, it determines which syntax run is the last one modified in the process. It then designates the offset of the character following the end of that run as the point up to which text must be redrawn in order to correctly update the display to reflect the changes made. This is to avoid redrawing any more than necessary, which can cause flicker and slow typing.

Before BBEdit actually does the redrawing based on this offset, it will make one more call to the language module with a `kBBLMAdjustEndMessage` to give the language module an opportunity to adjust the offset, if it wants to.

A `kBBLMAdjustEndMessage` call will be accompanied by the following parameter:

```
typedef struct
{
      SInt32 fEndOffset;
}
bblmAdjustEndParams;
```

You should naturally ensure that `fEndOffset` doesn't become less than zero or greater than the length of the document.

It is probably unlikely that you will ever need to provide a `kBBLMAdjustEndMessage` handler that does anything other than immediately return `noErr`. Only the TeX language module ever takes advantage of it. The TeX module divides a document up into runs of only two kinds: comments and everything else. The everything else runs tend to dominate and be quite large, and thus typing into them could become prohibitively slow if it had to redraw the entire run on every keystroke.

Because of the simple structure of the run list, the TeX module can safely assert that it is unnecessary to redraw text beyond the insertion point as long as the editing operation affected only a single run and rescanning did not alter the `language`, `kind`, or `startPos` of the run. If that is the case, the TeX `kBBLMAdjustEndMessage` handler gets the offset of the insertion point via the `bbxt_callbacks` and then passes that back to `fEndOffset`.

Advanced Topics

A Mystery Unshrouded

As mentioned earlier, the `bblmAdjustRangeParams` fields `fOrigStartIndex` and `fOrigStartRun` are included primarily for the benefit of the TeX language module and are not likely to be of interest to anyone else. Further on, it was also said that as long as the editing operation affected only a single run and rescanning did not alter the `language`, `kind`, or `startPos` of the run, the TeX module can safely assert that it is unnecessary to redraw text beyond the insertion point.

These two points are not unrelated. The TeX module uses the information given by `fOrigStartIndex` and `fOrigStartRun` to decide whether the conditions have been met to limit the extent of redrawing.

`fOrigStartIndex` is the value of `fStartIndex` that BBEdit begins with before it is possibly pushed backwards for language-independent reasons. `fOrigStartIndex` is only meaningful if the editing operation (that is to say the selection before the edit) is entirely contained within the run at that index. If this is not the case, then `fOrigStartIndex` is set to -1. Otherwise, `fOrigStartRun` is filled with the contents of the run, with its length adjusted to account for the number of characters inserted and/or deleted by the editing operation.

Module-defined Run Kinds

In some situations, it may be desirable for your language module to define run kinds for its own internal use. It may also use the predefined run kinds, and it may certainly use a combination of the two. The predefined run kinds are mapped to predefined color codes automatically by BBEdit.

If your module defines run kinds of its own, it needs to have a handler prepared to respond to a `kBBLMMapRunKindToColorCodeMessage`, whose parameters are as follows:

```
typedef struct
{
      SInt16   fRunKind;
      SInt16   fColorCode;
      Boolean  fMapped;
} bblmMapRunParams;
```

If your language module supports the given `fRunKind`, it should put in the corresponding `fColorCode`, set `fMapped` to `true`, and return `noErr`. If `fMapped` is not set to `true` or if you return anything other than `noErr`, `fColorCode` will be ignored and `kBBLMTextColor` will be used.

Module-defined Color Codes

There are some predefined color codes, in addition to the ones mentioned in "Generating Syntax Runs", that are used by the built-in HTML language module.

```
kBBLMSGMLTagColor
kBBLMSGMLAnchorTagColor
kBBLMSGMLImageTagColor
kBBLMSGMLAttributeNameColor
kBBLMSGMLAttributeValueColor
kBBLMXMLProcessingInstructionColor
```

As with run kinds, BBEdit automatically maps all predefined color codes to RGB colors. Your language module may define color codes of its own and, if it does so, it needs to have a handler prepared to respond to a `kBBLMMapColorCodeToColorMessage`, whose parameters are as follows:

```
typedef struct
{
      SInt16   fColorCode;
      RGBColor fRGBColor;
      Boolean  fMapped;
} bblmMapColorParams;
```

If your language module supports the given `fColorCode`, it should put in the corresponding `fRGBColor`, set `fMapped` to `true`, and return `noErr`. If `fMapped` is not set to `true` or if you return anything other than `noErr`, `fRGBColor` will be ignored and the color that `kBBLMTextColor` maps to will be used.

Note: The colors associated with all of the predefined color codes may be modified by the user via the BBEdit Preferences window. Unfortunately, there is at present no way for a language module to interact with the user for the purpose of selecting color mappings.

Coloring Keywords

Although it would have been possible to deal with language keywords as separate run kinds, doing so would result in so many transitions from plain text runs to keyword runs and back again that the size of run lists would quickly grow out of control, with undesirable performance and usability consequences. To avoid this pitfall, the code in BBEdit that draws runs of text takes special action when a run kind is mapped to `kBBLMTextColor`: it scans the text run as it draws, and colors keywords using the `kBBLMKeywordColor` instead.

Finding a keyword involves two steps that are under the control of the language module in effect for the text being scanned (that is to say, the module for the language of the run). The first is to detect the beginning and end of a possible keyword, according to whatever lexical rules apply in the given language, and the second is to determine whether the possible keyword is or is not, in fact, a keyword.

The former is accomplished via a simple table lookup. The table is preconfigured to recognize a fairly standard definition of a keyword: any combination of letters, digits, and the underscore character. If that fits your language's definition of the lexical syntax of a keyword, then no further action is required on your part. However, if you wish to include additional characters in the definition of a keyword, or exclude specific characters from the then your language module must include a handler for the `kBBLMSetCategoriesMessage`, whose parameters are as follows:

```
typedef struct
{
      BBLMCategoryTable fCategoryTable;
```

```
} bblmCategoryParams;
```

The `fCategoryTable` is a table of 256 8-bit characters whose entries should contain either of the following values:

`'a'` which indicates that the character associated with this entry is considered part of a "word" and thus may be part of a keyword.

`'-'` which indicates that the character associated with this entry is considered a "word break" character, i.e., one that falls between words.

Because the table only contains 256 entries, its applicability to non-Roman character encodings is limited. In fact, it is probably safest not to alter any entries associated with characters not in the 7-bit ASCII character set. The upper 128 entries are all preconfigured to contain `'-'` and any Unicode character whose value is greater than 255 is implicitly treated as though it had a `'-'` table entry.

If present, your `kBBLMSetCategoriesMessage` handler should modify the table (or not) as it deems appropriate and return `noErr`.

When BBEdit is scanning for keywords and encounters a "word" character, it collects all consecutive word characters into a string buffer. If your language module supplied a keyword-list in the `BBLMKeywordList`, BBEdit searches that list for a match.

If no match is found in the keyword table (or none is supplied), BBEdit calls the language module with a `kBBLMMatchKeywordMessage`, whose parameters are as follows:

```
typedef struct
{
        SInt8           *fToken;
        SInt16          fTokenLength;
        Boolean         fKeywordMatched;
} bblmKeywordParams;
```

If your handler determines that the `fTokenLength` characters at `fToken` constitute a keyword, it should set `fKeywordMatched` to true and return `noErr`.