HeaderDoc User Guide

A User's Guide to Self-Documenting Code

Developer

Contents

Tables and Listings

Introduction

This document describes how to use the HeaderDoc tool. It also explains how to insert HeaderDoc comments into your headers and other files. This document corresponds with HeaderDoc 8.0. For information about previous versions, consult the documentation installed with your HeaderDoc distribution.

What is HeaderDoc?

HeaderDoc is a set of tools for embedding structured comments in source code and header files written in various languages and subsequently producing rich HTML and XML output from those comments. HeaderDoc comments are similar in appearance to JavaDoc comments in a Java source file, but traditional HeaderDoc comments provide a slightly more formal tag set to allow greater control over HeaderDoc behavior.

HeaderDoc is primarily intended for use on OS X, as part of the OS X Developer Tools. However, in various versions, it has also been used successfully on other operating systems, including Linux, Solaris, and Mac OS 9. (Your mileage may vary.)

In addition to traditional HeaderDoc markup, HeaderDoc 8 supports JavaDoc markup. HeaderDoc 8 supports a wide range of languages:

- AppleScript
- Bourne shell (and Korn and Bourne Again)
- C Headers and C source code
- C++ headers
- C shell scripts
- Java
- JavaScript
- Mach MIG definitions
- Objective C/C++ headers
- Pascal
- Perl
- Python

- PHP
- Ruby
- Tcl

Also included with the main script (`headerdoc2html`) is `gatherheaderdoc`, a utility script that creates a master table of contents for all documentation generated by `headerdoc2html`. Information on running `gatherheaderdoc` is provided in "Advanced HeaderDoc Configuration and Features" (page 62).

Both scripts are typically installed in `/usr/bin`, as `headerdoc2html` and `gatherheaderdoc`.

The `gatherheaderdoc` script also uses a tool called `resolveLinks` to create links between documents. Although you probably won't need to use this tool directly, you can do so if you need to link together multiple sets of documentation. This tool is described in "Using resolveLinks to Resolve Cross References" (page 99).

HeaderDoc comes with a series of tools for man page generation, `xml2man` and `hdxml2manxml`. The first tool, `xml2man`, converts an mdoc-like XML dialect into mdoc-style man pages. The second tool, `hdxml2manxml`, converts HeaderDoc XML (generated with the -X flag) into a series of .mxml files suitable for use with `xml2man`.

You should read this document if you are interested in generating documentation from your source code, generating manual pages, or using any of HeaderDoc's other features.

How Do I Get It?

HeaderDoc is available in two ways. First, HeaderDoc is part of the standard OS X Developer Tools installation. If you have installed the Developer Tools CD, it is already installed on your system.

Second, HeaderDoc can be downloaded from the Darwin source collection at http://www.opensource.apple.com/darwinsource/.

Organization of This Document

This document is divided into several chapters describing various aspects of the tool suite.

- "Using HeaderDoc" (page 10) explains the syntax for the HeaderDoc command-line tool itself.
- "HeaderDoc Tags" (page 16) explains how to add HeaderDoc markup to header (and source code) files.
- "Basic HeaderDoc Configuration" (page 54) explains the HeaderDoc configuration file.
- "Advanced HeaderDoc Configuration and Features" (page 62) explains how to use `gatherheaderdoc` to produce landing pages and cross-linked trees of related documentation.

- "Using the MPGL Suite" (page 72) explains how to use the Manual Page Generation Language (MPGL) tool suite.
- "HeaderDoc Release Notes" (page 86) provides recent version history for the HeaderDoc toolchain.
- "Symbol Markers for HTML-Based Documentation" (page 92) describes the symbol markers used by HeaderDoc and various other utilities to provide linking functionality.
- "HeaderDoc Class Hierarchy" (page 104) describes the class hierarchy of the HeaderDoc tool itself.
- "Troubleshooting" (page 107) explains common error messages and their likely causes.

Using HeaderDoc

HeaderDoc includes two scripts, `headerdoc2html` (`headerDoc2HTML.pl` in the source distribution), which generates documentation for each header it encounters, and `gatherheaderdoc` (`gatherHeaderDoc.pl` in the source distribution), which finds these islands of documentation and assembles a master table of contents linking them together.

The `gatherheaderdoc` tool is a postprocessing script for HeaderDoc. Its primary purpose is to take a directory containing output from HeaderDoc and create a table of contents with links.

The `gatherheaderdoc` tool is highly configurable. You can configure it to insert custom breadcrumb links, use a custom TOC template, and even automatically insert "framework" information into the TOC template, if desired.

This chapter is divided into three parts:

- "Running headerdoc2html" (page 10)—information about running `headerdoc2html`.
- "Running gatherheaderdoc" (page 15)—information about running `gatherheaderdoc`.
- "Cocoa Front End" (page 15)—information about the Cocoa front end.

Running headerdoc2html

Once you have a header containing HeaderDoc comments, you can run the `headerdoc2html` script to generate HTML output like this:

```
> headerdoc2html MyHeader.h
```

This will process `MyHeader.h` and create an output directory called `MyHeader` in the same directory as the input file. To view the results in your web browser, open the file `index.html` that you find inside the output directory.

Instead of specifying a single input file (as above), you can specify an input directory if you wish. HeaderDoc will process every `.h` file in the input directory (and all of its subdirectories), generating an output directory of HTML files for each header that contains HeaderDoc comments.

HeaderDoc and Object-Oriented Languages

HeaderDoc processes C++ and Objective-C headers in much the same way that it does a C header. In fact, until HeaderDoc encounters a class declaration in a C++ header, the processing is identical.

When HeaderDoc generates the HTML documentation for a C++ or Objective-C header, it creates one frameset for the header as a whole, and separate framesets for each class, protocol, or category declared within the header.

> **A Note About Objective-C Categories:** An Objective-C category lets you add methods to an existing class. When HeaderDoc processes a batch of headers and finds comments for methods declared in a category, it searches for the associated class documentation and adds those methods and their documentation to the class documentation. If the class is not present in the current batch, HeaderDoc will create a separate frameset of documentation for the category.

Within Objective-C class declarations, you can use the `@method` tag to document each method. Since Objective-C is a superset of C, the header might also declare types, functions, or other API outside of any class declaration. You would use `@typedef`, `@function`, and other C tags to document these declarations.

> **Note:** The `@method` tag will generate faulty markup if the enclosing class does not have HeaderDoc markup. If this occurs, you will receive a warning that says that the `@method` tag is being used outside a class. To correct this, add a HeaderDoc comment for the enclosing class.

HeaderDoc records the access control level (public, protected, or private) of API elements declared within a C++ class. This information is used to further group the API elements in the resulting documentation.

HeaderDoc Command-line Switches

HeaderDoc has a number of useful command-line switches that alter its behavior.

Flag	Description
`-C` `--class-as-composite`	Causes HeaderDoc to output class contents as a composite page instead o separate pages for functions, data types, and so on.
`-D` <*token*> `-D` <*token*>`=`<*value*> `--defined` <*token*> `--defined` <*token*>`=`<*value*>	Specifies that the token should be explicitly defined to the specified value (o for C preprocessing purposes.

Flag	Description
`-E` `--process-everything`	Process everything. With this flag, HeaderDoc attempts to process an entir that has no HeaderDoc markup. **Note:** Not all programming languages support this flag.
`-F` `--old-style-frames`	Tells HeaderDoc to generate framesets instead of using iframe elements. **Note:** In HeaderDoc 8.7, you should generally specify this flag because of a opening in a new page in the iframe-based output. See "Late-Breaking Bu info and patches.
`-H` `--insert-header`	Turns on inclusion of the `htmlHeader` line, as specified in the configuratio
`-L` `--suppress-local-variables`	Disables emission of documentation for function-local variables (documen inside a function's documentation block). This allows you to have a traditio documentation for public API purposes and a more complete version for i
`-M` `--man-section`	Specifies the section number to use when generating man page content w
`-N` `--ignore-all-names`	Ignore all names specified in HeaderDoc tags (except for anonymous enur provided by the code) and use the names specified by the code instead.
`-O` `--outer-names-only`	Enables "outer name only" type handling, in which tag names for typedefs example, `foo` in `typedef struct foo {...} tdname;`).
`-P` `--pipe-output`	Pipe mode. In this mode, HeaderDoc prints the resulting XML content (def standard output. You can only process a single file at a time in this mode.
`-Q` `--paranoid`	The opposite of quiet, this flag enables paranoid warnings about a numbe
`-S` `--merge-superclass-docs`	Causes HeaderDoc to include functions and data types from the superclas of child classes (if they are processed at once).
`-T` <*mode*> `--test` <*mode*>	HeaderDoc test mode. Note that this test mode is only available when runn not from the installed system. For more information, see "Testing HeaderD
`-U` <*token*> `--undefined` <*token*>	Specifies that the token should be explicitly undefined for C preprocessing

Flag	Description
`-X` `--xml-output`	Causes HeaderDoc to output XML content instead of HTML.
`-a` `--align-columns`	Tells HeaderDoc to attempt to align function parameters vertically after th instead of indenting them by a single tab width.
`-b` `--basic-processing-only`	Puts HeaderDoc into "basic" mode. In this mode, numbered lists are not au and embedded HeaderDoc comments are not removed from declarations.
`-c` `--config-file`	Allows you to add an alternate configuration file. For example: `headerdoc2html -c myCustomHeaderDocConfigFile.config MyHeade`
`-d` `--debugging`	Turns on extra debugging output.
`-e` `--exclude-list-file`	Specifies an exclude list. The file passed as an argument to the `-e` flag conta list of Perl regular expressions. Any filename or file path that matches any of t is excluded from processing.
`-f` `--function-list-output`	Enables function list output mode. In this mode, HeaderDoc emits a simple encountered and the contents of those functions in an easily machine-par
`-g` `--group-right-side`	Group content on the right side by group instead of alphabetically.
`-i` `--truncate-function-like-macros`	Tells HeaderDoc to output the body of macro declarations.
`-j` `--allow-javadoc-syntax`	Enables support for JavaDoc (`/**`) comment markers in other programmin
`-l` `--no-link-requests`	Tells HeaderDoc not to generate link requests in declarations.
`-m` `--man-page-output`	Tells HeaderDoc to generate a man page for each function found in lieu of g output.
`-n` `--ignore-apiowner-names`	Ignore the names of classes and headers specified in HeaderDoc tags and provided by the code itself.

Flag	Description
`-o` <*directory*> `--output-directory` <*directory*>	Allows you to specify another directory for the output. For example: `headerdoc2html -o /tmp MyHeader.h`
`-p` `--enable-cpp`	Enables the C preprocessor. With this switch, any `#define` with HeaderDo content that appears after it within the same header file, and also affects a `#include` in any file that includes that header file.
`-q` `--quiet`	Makes HeaderDoc operate silently (except for warnings and errors).
`-s` `--strip`	Causes HeaderDoc to enter a comment stripping mode, in which it output file in the output directory from which all HeaderDoc comments have bee
`-t` `--enforce-strict-tagging`	Enables strict tagging mode, in which any function parameters not describ result in a warning.
`-u` `--unsorted`	Disables sorting of functions, data types, and so on in the table of content original file order. Note that if you simply want to preserve groupings, you or `@functiongroup` tags instead.
`-v` `--version`	Tells HeaderDoc to print version information and exit.
`-x` `--doxytags`	Causes HeaderDoc to emit a Doxygen-style tag file. (Note that this variant o class inheritance information that is not typically included in a normal Dox
`--tocformat`	Sets the format used for the table of contents (left side). Valid values are: • `default`—use the most current TOC style (currently `div`). • `div`—use the div format (currently the default). • `frames`—use the legacy frames format. • `iframes`—use the legacy iframes format.
`--apple`	Enables various flags and additional policy checks specific to Apple interna
`--auto-availability`	Interpret `#if` and `#ifdef` blocks that contain availability information and the availability information. (This is probably not useful for projects that ar
`--document-internal`	Include documentation marked with `@internal`. By default, this docume

Most of these switches can be used in combination with each other. The obvious exceptions are `-X` and `-m` (XML vs. man page output). If you need both XML and man page output, you should specify the `-X` flag (XML output), then run the scripts `hdxml2manxml` and `xml2man` to convert the XML output to a man page yourself.

Running gatherheaderdoc

The `gatherheaderdoc` script scans an input directory (recursively) for any documentation generated by `headerdoc2html`. It creates a master table of contents (named `masterTOC.html` by default—the name can be changed by setting a new name in the configuration file or by specifying a second argument). It also adds a "top" link to all the documentation sets it visits to make it easier to navigate back to the master table of contents.

Here's an example of how to create documentation for a number of headers (the sample ones provided with the scripts) and then generate a master table of contents:

```
> headerdoc2html -o OutputDir ExampleHeaders
> gatherheaderdoc OutputDir
```

You can now open the file `OutputDir/masterTOC.html` in your browser to see the interlinked sets of documentation.

You can also add a second argument to change the output file name. For example:

```
> headerdoc2html -o OutputDir ExampleHeaders
> gatherheaderdoc OutputDir MYTOCNAME.html
```

This time, `gatherheaderdoc` created the file `OutputDir/MYTOCNAME.html` instead of `OutputDir/masterTOC.html`.

For more information on configuring `gatherheaderdoc`, see "Basic HeaderDoc Configuration" (page 54).

Cocoa Front End

Kyle Hammond has made a Cocoa front end available for HeaderDoc. OS X users can download this from http://kyle.snowmintcs.com/cocoa_programming.php.

HeaderDoc Tags

Tags, depending on type, generally require either one field of information or two:

- `@function` [FunctionName]
- `@param` [parameterName] [Some descriptive text...]

In the tables in this chapter, the "Fields" column indicates the number of textual fields each type of tag takes.

Introduction to HeaderDoc Comments and Tags

HeaderDoc comments in C and C-like languages (Java, JavaScript, IDL, PHP, and MIG) are of the form:

```
/*!
 This is a comment about FunctionName.
*/
char *FunctionName(int k);
```

In their simplest form (as above) they differ from standard C comments only by the addition of the `!` character next to the opening asterisk.

In AppleScript and Pascal, the syntax is almost identical except for the comment markers:

Listing 2-1 AppleScript comment example

```
(*! This is a comment about FunctionName. *)
on FunctionName(x,y)
    ...
end FunctionName
```

Listing 2-2 Pascal comment example

```
{! This is a comment about FunctionName. }
```

```
procedure FunctionName(a, b: String; j: Integer): Char;
begin
    ...
end;
```

In Perl, Tcl, and shell scripts, the syntax is slightly altered because of the lack of multi-line comments and the need to avoid conflicting with the shell magic (`#!`) syntax. Shell script and Perl script HeaderDoc comments look like this:

Listing 2-3 Perl comment example

```
# /*!
#     This is a comment about FunctionName.
#  */
sub FunctionName($$)
{
    ...
}
```

Listing 2-4 Shell comment example

```
# /*!
#     This is a comment about FunctionName.
#  */
FunctionName()
{
    ...
}
```

Similarly, Ruby and Python syntax do not lend themselves to a start-of-comment marker followed by a single exclamation point, so their comments look like this:

Listing 2-5 Ruby comment example

```
=begin
    !headerDoc!
    This is a comment about FunctionName.
```

```
=end
def FunctionName(arg)
    ...
end
```

Listing 2-6 Python comment example

```
"""
    !headerdoc!
    This is a comment about FunctionName.
"""
def FunctionName(x):
   ...
```

Historically, HeaderDoc tags were required to begin with an introductory tag that announces the type of API being commented (`@function`, below). You can find a complete list of these tags in "Top-Level HeaderDoc Tags" (page 38). Beginning in HeaderDoc 8, these top-level tags became optional. However, providing these tags can, in some cases, be used to cause HeaderDoc to document something in a different way. One example of this is the use of the `@class` tag to modify the markup of a typedef, as described in "Overriding the Default Data Type: C Pseudoclass Tags" (page 50).

The following example shows the historical syntax:

```
/*!
 @function FunctionName
 This is a comment about FunctionName.
*/
char *FunctionName(int k);
```

Following the optional top-level `@function` tag, you typically provide introductory information about the purpose of the class. You can divide this material into a summary sentence and in-depth discussion (using the `@abstract` and `@discussion` tags), or you can provided the material as an untagged block of text, as the examples below illustrate. You can also add `@throws` tags to indicate that the class throws exceptions or add an `@namespace` tag to indicate the namespace in which the class resides.

Listing 2-7 Example of documentation with @abstract and @discussion tags

```
/*!
     @class IOCommandGate
     @abstract Single-threaded work-loop client request mechanism.
     @discussion An IOCommandGate instance is an extremely light weight mechanism
 that
         executes an action on the driver's work-loop...
     @throws foo_exception
     @throws bar_exception
     @namespace I/O Kit (this is just a string)
     @updated 2003-03-15
 */
class IOCommandGate: public IOEventSource
{
...
}
```

Listing 2-8 Example of documentation as a single block of text

```
/*!
     @class IOCommandGate
     A class that defines a single-threaded work-loop client request mechanism.
An IOCommandGate
     instance is an extremely light weight mechanism that executes an action on
the driver's work-loop...
     @abstract Single-threaded work-loop client request mechanism.
     @throws foo_exception
     @throws bar_exception
     @updated 2003-03-15
*/
class IOCommandGate: public IOEventSource
{
...
}
```

Note: Once you have specified a non-inline tag such as @abstract, that tag is active until the next non-inline tag. This means that general discussion paragraphs can only occur in one of four places:

- At the beginning of the comment.
- Immediately following an introductory top-level tag such as `@class`.
- Immediately following a discussion tag (`@discussion`).
- After an empty line in an `@brief` tag (which is identical to `@abstract` except that it stops at the first blank line).

You can also use additional JavaDoc-like tags within the HeaderDoc comment to identify specific fields of information. These tags will make the comments more amenable to conversion to HTML. For example, a more complete comment might look like this:

```
/*!
     @function HMBalloonRect
     @abstract Reports size and location of help balloon.
     @discussion Use HMBalloonRect to get information about the size of a help
balloon
         before the Help Manager displays it.
     @param inMessage
        The help message for the help balloon.
     @param outRect
        The coordinates of the rectangle that encloses the help message.
        The upper-left corner of the rectangle has the coordinates (0,0).
 */
```

Tags are indicated by the @ character, which must generally appear as the first non-whitespace character on a line (with a few notable exceptions). If you need to include an at sign in the output (to put your email address in a class discussion, for example), you can do this by prefixing it with a backslash, that is, \@. (If you forget the backslash, to the extent that it is possible to do so, HeaderDoc ignores unknown tags and treats them as though you had quoted the @ characters, but it does produce a warning. Because new tags are periodically added, you should not count on this behavior.)

The first tag in a comment announces the API type of the declaration (`function`, `struct`, `enum`, and so on). This tag is optional. If you leave it out, HeaderDoc will pick up this information from the declaration immediately following the comment.

The next two lines (tagged `@abstract` and `@discussion`) provide documentation about the API element as a whole. The abstract can be used in summary lists, and the discussion can be used in the detailed documentation about the API element.

The abstract and discussion tags are optional, but encouraged. Their use enables various improvements in the HTML output, such as summary pages. However, if there is untagged text following the API type tag and name (`@function HMBalloonRect`, above) it is assumed to be a discussion. With such untagged text, HeaderDoc assumes that the discussion extends from the end of the API type tag to the next HeaderDoc tag or to the end of the HeaderDoc comment, whichever occurs first.

HeaderDoc understands some variants in commenting style. In particular, you can have a one-line comment like this:

```
/*! @var settle_time Latency before next read. */
```

Be sure, however, to understand the difference between the above syntax and the following, which is treated as a multiword name (described in "Multiword Names" (page 22)):

```
/*! @enum my favorite enumeration
    This is the discussion here.
 */
```

You can also use leading asterisks on each line of a multiline comment (but you must use them consistently):

```
/*!
 * @function HMBalloonRect
 * @abstract Reports size and location of help ballon.
 * @discussion Use HMBalloonRect to get information about the size of a help balloon
 * before the Help Manager displays it.
```

```
 * @param inMessage The help message for the help balloon.
 * @param outRect The coordinates of the rectangle that encloses the help message.
 * The upper-left corner of the rectangle has the coordinates (0,0).
 */
```

If you want to specify a line break in the HTML version of a comment, use two newline characters between lines rather than one. For example, the text of the discussion in this comment:

```
/*!
 * @function HMBalloonRect
 * @discussion Use HMBalloonRect to get information about the size of a help balloon
 * before the Help Manager displays it.
 *
 * Always check the help balloon size before display.
 */
```

will be formatted as two paragraphs in the HTML output:

HMBalloonRect

```
OSErr HMBalloonRect (const HMMessageRecord *inMessage, Rect *outRect);
```

Use HMBalloonRect to get information about the size of a help balloon before the Help Manager displays it.

Always check the help balloon size before display.

Multiword Names

Top-level HeaderDoc tags, such as `@header` and `@function` can take multiword names. This is particularly useful for documenting anonymous types for enumerations, for example. However, HeaderDoc normally has no way to know whether a line containing multiple words is a multiword name or a name followed by a discussion.

There are two ways to get a multiword name. One way is to add an explicit discussion tag (which may be empty), like this:

Listing 2-9 Example of multiword names using @discussion

```
/*!
 * @enum example enum
 * @discussion This is a test, this is only a test.
 *
 * Because we included an \@discussion tag, the name of the enum is
 * "example enum".
 */
```

The other way is to simply add a line break and additional discussion (which may *not* be empty) after the name.

Listing 2-10 Example of multiword names using multiple lines

```
/*!
 * @enum example enum
 * Because the discussion continues on a second line,
 * the name of the enum is "example enum".
 */
```

Automatic Tagging

Beginning in HeaderDoc 8, certain tags are often not needed. These include:

Numbered lists

It is no longer necessary to mark up numbered lists with `<ol><li>`. HeaderDoc will automatically detect numbered lists.

Declaration types

Declaration type tags such as `@function`, `@class`, and `@typedef` are no longer required unless you are trying to override HeaderDoc's normal behavior (such as using `@class` or `@interface` to change the display of a `typedef struct`.

Availability macros

It is no longer necessary to ignore availability macros with `@ignore`. The file `Availability.list` in the HeaderDoc modules directory contains a mapping of availability macros to strings. When any macros described in this file appear in a declaration, the corresponding text will automatically be added to its documentation as an availability attribute.

You can add your own availability macros by adding them to the `Availability.list` file or by adding an `@availabilitymacro` block in your headers.

Types of Tags

HeaderDoc tags can be grouped into two broad categories: top-level tags and second-level tags. Top-level tags describe the type of declaration that follows. For example, `@method` indicates that the following declaration is a method. All other tags are second-level tags.

Top-Level Tags

Top-level HeaderDoc tags tell HeaderDoc what API type to expect after the declaration. These trace their roots back to HeaderDoc 7 and prior releases in which HeaderDoc could not interpret a declaration without these hints.

In HeaderDoc 8 and later, top-level tags are almost always optional (except for tags that are not tied to a declaration, such as `@header` or `@group`). Some top-level tags provide useful features, however—declaring new availability macros, treating one type of declaration as another type, and so on.

Most top-level HeaderDoc tags are treated as a term and definition list. This means that if you specify multiple words on a single line, the first word is treated as the name, and the remaining words are treated as the discussion. However, if the arguments span multiple lines, the entire first line is treated as a multi-word title. Similarly, if you specify an `@discussion` tag explicitly, the entire first line is treated as a multi-word title. For more information, see "Multiword Names" (page 22).

Group tags (`@functiongroup`, `@group`, and `@methodgroup`) always treat the remainder of the line as a multi-word name.

If you include a top-level tag, it *must* appear at the beginning of the HeaderDoc comment. These tags represent declaration types. For example, the `@function` tag tells HeaderDoc that you are about to declare a function. These tags are optional, and are generally discouraged because they tend to cause frequent mistakes.

Second-Level Tags

Second-level tags give HeaderDoc additional information about the declaration, such as specifying an abstract or a parameter description.

The set of second-level tags can be further divided up based on their behavior:

- **attribute**—A tag whose contents appear as an line in a table or list of attributes. Attribute tags continue until the next block or attribute tag; however, you should generally keep their contents short.
- **block**—A tag that can contain multiple paragraphs of text and is usually displayed as a normal block of text (often prefaced by a heading). Block tags continue until the next block or attribute tag.
- **flag**—A tag that modifies the behavior of a tagged declaration, including whether or not to emit it under certain circumstances (`@parseOnly`, for example). Flag tags take no arguments.
- **HTML tagging**—A tag that affects HTML tagging and is not directly emitted as part of the output.
- **inline**—A tag that can appear within a paragraph inside most tags (except for name or title fields). The contents of an inline tag do not break the text flow.
- **page footer**—A tag that modifies content that appears in the footer at the bottom of each content page (`@copyright`, for example).
- **parsing**—A tag that modifies the way the source code file is parsed.
- **term & definition**—A tag whose contents get split into two parts at the first space or newline, depending on whether the tag contains one or more lines of content. These tags are split according to the same rules as top-level tags. The splitting rules are described in "Multiword Names" (page 22)

In HeaderDoc 8.6 and later, second-level tags can appear anywhere in a HeaderDoc comment. (In HeaderDoc 8.5 and earlier, comments must begin with either a top-level tag or an untagged discussion.)

There are three exceptions, however: the `@const`, `@constant`, and `@var` tags. These tags cannot appear at the beginning of a HeaderDoc comment because they conflict with top-level tags.

HeaderDoc Tags Common to All Comment Types

The tags in the table below can be used in any comment for any data type, function, header, or class.

Table 2-1

Tag	Example	Identifies	Usage
`@abstract`	`@abstract write the track to disk`	A short string that briefly describes a function, data type, and so on. This should not contain multiple lines (because it will look odd in the mini-TOCs). Save the detailed descriptions for the discussion tag.	block (single short sen recommended)
`@apiuid`	`@apiuid //my_ref/doc/magic`	Overrides the API UID (`apple_ref`) associated with this comment. Note that very little checking is performed on this string. Thus, this tag has the potential to seriously break your output if used incorrectly. It is primarily provided for resolving symbol collisions that are otherwise unavoidable, and is generally discouraged.	attribute Must contain no and must be a va reference. See "S Markers for HTML-Based Documentation" 92) for details.

`@attribute`
`@attributelist`
`@attributeblock`
	See "Adding Arbitrary Attributes" (page 53).	Adds arbitrary attributes.	attribute (short, definition list, or

`@availability`	`@availability 10.3 and later`	A string that describes the availability of a function, class, and so on.	attribute
`@brief`	`@brief write the track to disk`	Equivalent to @abstract. Provided for better Doxygen compatibility.	block (single short sen recommended)

Tag	Example	Identifies	Usage
`@discussion`	`@discussion This is what this function does. @some_other_tag`	A block of text that describes a function, class, header, or data type in detail. This may contain multiple paragraphs. `@discussion` may be omitted, as described above. `@discussion` must be present if you have a multiword name for a data type, function, class, or header. An `@discussion` block ends only when another block begins, such as an `@param` tag.	block
`@indexgroup`	`@indexgroup Name of Group`	Provides grouping information within the master TOC (landing page). In the absence of an @indexgroup tag, the index group is inherited from the enclosing class or header.	block (short strin please)
`@internal`	`@internal`	Marks the declaration as internal documentation. Such documentation is emitted only if the `--document-internal` flag is specified on the command line. **Note:** The declaration may still modify other declarations in the case of `#define` macros with C preprocessing enabled.	Flag (takes no arguments).

Tag	Example	Identifies	Usage

`@link`	`@link //apple_ref/c/func/function_name link text goes here @/link`
	or
	`@link function_name link text goes here @/link`
	or
	`@link function_name @/link`
		Allows you to insert a link request for an API ref. See "Creating Links Between Symbols" (page 51) for more information.	inline

`@namespace`	`@namespace BSD Kernel`	String describing the namespace in which the function, data type, etc. exists.	attribute
`@see`	`@see apple_ref Title for link`	Adds a "See:" entry to the attributes. Arguments are the same as `@link`. Note that this tag is ignored if the API reference marker already appears in the see or see also list.	attribute
`@seealso`	`@seealso apple_ref Title for link`	Adds a "See Also:" entry to the attributes. Arguments are the same as `@link`. Note that this tag is ignored if the API reference marker already appears in the see or see also list.	attribute
`@textblock`	`@textblock My text goes here @/textblock`	Treat everything until the trailing `@/textblock` as raw text, preserving initial spaces and line breaks, and converting "<" and ">" to "<" and ">". **Note:** This tag does not automatically insert `<pre>` or `<tt>`. You may wrap it with whatever formatting you choose.	inline

Tag	Example	Identifies	Usage
`@updated`	`@updated 2003-03-14`	The date at which the header was last updated.	attribute

In addition, HeaderDoc supports some common JavaDoc and Doxygen synonyms for other tags: `@since` (`@availability`), `@details` (`@discussion`), and `@description` (`@discussion`)

HeaderDoc Comment Types

HeaderDoc handles comments differently based on the declaration that follows them. Although most tags are valid in any HeaderDoc comment, there are a few tags that only make sense in certain contexts. For example, a method can have parameters, but a class cannot.

This section describes each type of HeaderDoc comment, and provides a list of second-level tags that are specific to that comment type.

Frameworks or Other Doc Groupings

Top-level tag: `@framework`

The `@framework` tag is used to describe a set of related headers. You should put this framework documentation into a file whose name ends with `.hdoc`. When you run `headerdoc2html` on such a file, it generates a documentation tree with special hidden markup that `gatherheaderdoc` then parses and uses while generating the landing page (master TOC).

Frameworks tags must contain an `@framework` tag that provides a human-readable (long) name for the framework that `gatherheaderdoc` inserts wherever the `$$framework@@` tag appears in the landing page template.

The following tags are specific to frameworks:

Table 2-2

Tag	Example	Identifies	Ty
`@frameworkcopyright`	`@frameworkcopyright 2010 Somebody Else.`	The copyright info for the header.	at
`@frameworkpath`	`@frameworkpath /System/Library/Frameworks/Kernel.framework`	The path to the framework.	at

Tag	Example	Identifies	Ty
`@frameworkuid`	`@frameworkuid myCustomUID`	Specifies a unique ID that `gatherHeaderDoc` inserts as part of a special anchor when building the main TOC. (This is not particularly useful externally, but is included for completeness.)	at
`@headerpath`	`@headerpath /blah/blah/blah.framework/Headers`	Provides the path to the `Headers` folder inside the framework.	at

For example:

```
/*! @framework Kernel Framework Reference
    @abstract
    @discussion The Kernel Framework provides the APIs and support for
    kernel-resident device drivers and other kernel extensions.
    It defines the base class for I/O Kit device drivers (IOService),
    several helper classes, and the families supporting many types
    of devices.
    @frameworkpath /System/Library/Frameworks/Kernel.framework
    @frameworkuid TP30000816
    @seealso //apple_ref/doc/uid/TP0000011 I/O Kit Fundamentals
    @seealso //apple_ref/doc/uid/TP30000905-CH204 Kernel Programming
 */
```

Headers and Source Files

Top-level tags: `@header`, `@file`

Often, you'll want to add a comment for the header as a whole in addition to comments for individual API elements. For example, if the header declares API for a specific manager (in Mac OS terminology), you may want to provide introductory information about the manager or discuss issues that apply to many of the functions within the manager's API. Likewise, if the header declares a C++ class, you could discuss the class in relation to its superclass or subclasses.

In general, you should not specify a filename in the `@header` tag. However, if you do, the value you give for the `@header` tag serves as the title for the HTML pages generated by `headerdoc2html` (unless you pass the `–n` or `–N` flag, in which case it is ignored).

The discussion for the `@header` tag is used as the introduction.

Note that you must follow `@header` by a line break; otherwise, the first line of your documentation will be treated as if it were the name of the header.

The following tags are specific to header and source file comments:

Tag	Example	Identifies	Type
`@author`	`@author Anon E. Mouse`	The author of the header.	attribute
`@charset`	`@charset utf-8`	Sets the character encoding for generated HTML files (same as `@encoding`).	HTML tagging
`@compilerflag`	`@compilerflag -lssl`	Compiler flag that should be set when using functions and types in this header.	attribute (term & definition)
`@copyright`	`@copyright Apple`	Copyright info to be added to each page. This overrides the config file value and may not span multiple lines.	page footer

Tag	Example	Identifies	Type
`@CFBundleIdentifier`	`@CFBundleIdentifier org.mklinux.driver.test`	Which kernel subcomponent, loadable extension, or application bundle contains this header	attribute
`@encoding`	`@encoding utf-8`	Sets the character encoding for generated HTML files (same as `@charset`).	HTML Tagging
`@flag`	`@flag -lssl` `The SSL Library`	Same as `@compilerflag`.	attribute (term & definition)
`@ignore`	`@ignore API_EXPORT`	Tells HeaderDoc to delete the specified token.	parsing
`@ignorefuncmacro`	`@ignorefuncmacro __P`	Tells HeaderDoc to unwrap occurrences of the specified function-like macro.	parsing
`@language`	`@language c++`	**Deprecated.** Sets the current programming language.	parsing

Tag	Example	Identifies	Type

`@meta`	`@meta robots index,nofollow`
	or
	`@meta http-equiv="refresh" content="0;http://www.apple.com"`
		Meta tag info to be added to each page. This can be either in the form `@meta <name> <content>` or `@meta <complete tag contents>`, and may not span multiple lines.	HTML tagging

`@preprocinfo`	`@preprocinfo This header uses the DEBUG macro to enable additional debugging.`	Description of behavior when preprocessor macros are set (`-DDEBUG`, for example).	block
`@related`	`@related Sixth cousin of Kevin Bacon.`	Indicates another header that is related to this one. You may use multiple `@related` tags. Similar to the `@seealso` tag.	attribute (term & definition)
`@unsorted`	`@unsorted`	Indicates that you do not want HeaderDoc to alphabetize the contents of this header.	flag
`@version`	`@version 2.3.1`	the version number to which this documentation applies.	attribute
`@whyinclude`	`@whyinclude Because it was there.`	Indicates why you should include the header.	attribute

Listing 2-11 Example of `@header` tag

```
/*!
     @header Repast Manager
     The Repast Manager provides a functional interface to the repast driver.
     Use the functions declared here to generate, distribute, and consume meals.
     @copyright Dave's Burger Palace
     @updated 2003-03-14
     @meta http-equiv="refresh" content="0;http://www.apple.com"
 */
```

Classes, Interfaces, Protocols, and Packages

Top-level tags: `@class`, `@interface`, `@protocol`, `@category`, `@template`

HeaderDoc supports a number of tags specific to classes, interfaces, protocols, and packages:

Tag	Example	Identifies	Type
`@classdesign`	`@classdesign Multiple paragraphs go here.`	Description of any common design considerations that apply to this class, such as consistent ways of handling locking or threading.	block in class declarations only
`@coclass`	`@coclass myCoClass Description of how class is used`	Class with which this class was designed to work.	attribute (term & definition) in class declarations only
`@dependency`	`@dependency This depends on the FooFramework framework.`	External resource that this class depends on (such as a class or file).	attribute in class declarations only

Tag	Example	Identifies	Type
`@deprecated`	`@deprecated in version 10.4`	String telling when the function, data type, etc. was deprecated.	attribute in class declarations only
`@helper` or `@helperclass`	`@helper myHelperClass` `Description of how class is used.`	A helper class used by this class.	attribute (term & definition) in class declarations only
`@helps`	`@helps This class provides additional stuff that does something.`	If this is a helper class, a short description of classes that this class was designed to help.	attribute in class declarations only
`@instancesize`	`@instancesize Eight hundred megabytes and constantly swapping.`	The typical size of each instance of the class.	attribute in class declarations only
`@ownership`	`@ownership MyClass objects are owned by the MyCreatorClass object that created them.`	Describes the ownership model to which this class conforms.	block in class declarations only
`@performance`	`@performance This class is strongly discouraged in high-performance contexts.`	Describes special performance characteristics for this class.	block in class declarations only
`@security`	`@security This class is feeling insecure today.`	Describes security considerations associated with the use of this class	block in class and header declarations only

Tag	Example	Identifies	Type
`@superclass`	`@superclass fasterThanASpeedingRuntime`	Overrides superclass name—see note below.	attribute in class declarations only
`@templatefield`	`@templatefield base_type` `The base type to store in the linked list.`	Each of the function's template fields (C++).	attribute (term & definition) in C++ class declarations only
`@unsorted`	`@unsorted`	Indicates that you do not want HeaderDoc to alphabetize the contents of this class.	flag
`@var`	`@var myVar` `Description goes here`	Used to document an instance variable in Perl. **Note:** Because this tag has the same name as a top-level tag, it cannot be the first tag in the HeaderDoc comment for a class.	Term & definition Valid only for Perl packages.

Note: The `@superclass` tag is not generally required for superclass information to be included. The `@superclass` tag has two purposes:

• To add "superclass" info to a C pseudo-classes such as a COM interface (a `typedef struct` containing function pointers).

• To enable inclusion of superclass functions, types, etc. in the subclass docs. The superclass *MUST* be processed before the subclass (earlier on the command line or higher up in the same file), or this may not work correctly.

Objective-C Classes, Protocols, and Interfaces

Here are some examples of classes in Objective-C:

Listing 2-12 Example of @class tag in Objective-C

```
/*!
     @class myInterface
     @discussion This is a discussion.
     It can span many lines or paragraphs.
*/
@interface myInterface : NSObject
@end
```

Listing 2-13 Example of @protocol tag in Objective-C

```
/*!
     @protocol myProtocol
     @discussion This is a discussion.
     It can span many lines or paragraphs.
*/
@protocol myProtocol
@end
```

Listing 2-14 Example of @category tag in Objective-C

```
/*!
 @category myCategory(myMainClass)
 @discussion This is a discussion.
 It can span many lines or paragraphs.
*/
@interface myCategory(myMainClass)
@end
```

C++ Classes

Listing 2-15 Example of @class tag in C++

```
/*!
```

```
 @class myClass
 @discussion This is a discussion.
 It can span many lines or paragraphs.
*/
 class myClass : public mySuperClass;
```

Listing 2-16 Example of @templatefield tag

```
/*! @class mystackclass
    @templatefield Tthe data type stored in this stack */

 template <T> class mystackclass
```

Groups of Declarations

Top-level tags: `@functiongroup`, `@methodgroup`, `@group`

Grouping tags allow you to organize functions, methods, and variables into collections. In HTML output mode, the table of contents (left column) is organized into these groups. Also, the body content (right side) contains a documentation for each group. That documentation block contains the group's name, discussion, and a list of any functions, data types, or variables contained within that group, along with their abstracts.

The `@group` tag provides grouping for all API elements except for methods and functions. In addition, until HeaderDoc encounters an `@functiongroup` or `@methodgroup` tag, functions and methods are also grouped by the `@group` tag. In effect, the `@functiongroup` or `@methodgroup` tag provides a way to override the `@group` tag in a way that only affects functions and methods.

Grouping tags remain in effect until the next grouping tag of the same type.

> **Note:** The `@functiongroup` and `@methodgroup` tags modify the groupings for *both* functions and methods. The two names are provided strictly for naming consistency; both tags behave identically.

If you need to put functions or other API elements in different parts of the header into the same group, simply give them the same name (with the same capitalization, punctuation, spacing, etc.), and HeaderDoc merges the two function groups into one. (Omit the discussion after the first occurrence.)

Any functions or other API elements encountered before the first `@group` or `@functiongroup` are considered part of the "empty" group. These functions are listed before any grouped functions or API elements.

Listing 2-17 Example of group tags

```
/*!
 @functiongroup Core Functions
*/
/*!
 @methodgroup Core Methods
*/
/*!
 @group Core API
*/
```

Functions, Methods, and Callbacks

Top-level tags: `@function`, `@method`, `@callback`

> **Note:** If you use a top-level tag, use the `@function` tag for C functions, and the `@method` tag for Objective-C methods. In all other languages, `@function` and `@method` are interchangeable.

Functions, methods, and callbacks have a number of special second-level tags:

Tag	Example	Identifies	Type
`@param`	`@param myValue The value to process.`	The name and description of a parameter to a function or callback.	attribute (term & definition)
`@result`	`@result Returns 1 on success, 0 on failure..`	Describes the return values expected from this function. Don't include if the return value is `void` or `OSERR`.	attribute (term & definition)
`@return`	`@return Returns 1 on success, 0 on failure..`	Same as `@result`.	attribute (term & definition)

Tag	Example	Identifies	Type
`@templatefield`	`@templatefield base_type The base type to store in the linked list.`	Each of the function's template fields (C++).	attribute (term & definition) in C++ method declarations only
`@throws`	`@throws bananas`	Include one `@throws` tag for each exception thrown by this function (in languages that support exceptions).	attribute
`@var`	`@var myVar` `Description goes here`	Documents a local variable in a function or method. You can suppress local variables in the output by passing the `–L` flag to `headerdoc2html`. **Note:** Because this tag has the same name as a top-level tag, it cannot be the first tag in the HeaderDoc comment for a function or method.	Term & definition

Listing 2-18 C function example

```
/*!
    This is a function.
    @param parmA
        Parameter A.
    @param parmB
        Parameter B.
    @result
        Returns something unexpected.
 */
SpanishInquisition *myFunction(char *parmA, int parmB);
```

Listing 2-19 Objective-C method example

```
/*!
    This is an objective-C method.
    @param parmA
        Parameter A.
    @param parmB
        Parameter B.
    @result
        Results in global warming.
 */
- (CO2 *)doSomething:(typeName)parmA withSomething:(typeName)parmB;
```

Constants and Variables

Top-level tag: `@var`, `@const`, `@constant`

The `@var` tag should be used when marking up global variables, class variables, and instance variables (as opposed to declarations of new data types or macros).

Variables that are immutable (`const` in C, for example) should be marked with `@const` or `@constant`.

Variable and constant declarations have no special second-level tags associated with them.

Listing 2-20 Example of @const tag

```
 /*!
     @const kCFTypeArrayCallBacks
    @abstract Predefined CFArrayCallBacks structure containing a set of callbacks
 appropriate...
     @discussion Extended discussion goes here.
         Lorem ipsum....
 */
const CFArrayCallBacks kCFTypeArrayCallBacks;
```

Listing 2-21 Example of @var tag

```
/*!
     @var we_are_root
     @abstract Tells whether this device is the root power domain
```

```
    @discussion TRUE if this device is the root power domain.
        For more information on power domains....
 */

bool we_are_root;
```

Objective-C Properties

Top-level tag: `@property`

In Objective-C, a property is a special variable that also includes getter and setter methods. It supports any tag that is supported by `@method` or `@var`.

> **Note:** JavaScript properties should be marked up as ordinary variables.

Structures and Unions

Top-level tags: `@struct`, `@union`, `@typedef`

Structures, unions, and `typedef` declarations containing structures and unions can contain callbacks and fields. The corresponding second-level tags are described below.

> **Note:** Although COM interfaces are special `typedef` declarations, HeaderDoc treats then as classes. You should mark up any fields or functions within a COM interface just as you would within a C++ class, rather than documenting them within the HeaderDoc comment block for the COM interface itself.

Tag	Example	Identifies	Type
`@callback`	`@callback testFunc The test function to call.`	Specifies the name and description of a callback field in a structure.	attribute (term & definition)
`@field`	`@field isOpen Specifies whether the file descriptor is open.`	A field in a structure declaration.	attribute (term & definition)

Listing 2-22 Example of a structure

```
/*!
     @struct TableOrigin
     @abstract Locates lower-left corner of table in screen coordinates.
     @field x Point on horizontal axis.
     @field y Point on vertical axis
     @discussion Extended discussion goes here.
         Lorem ipsum....
*/
struct TableOrigin {
 int x;
 int y;
}
```

Enumerations

Top-level tags: `@enum`, `@typedef`

The only tag specific to enumerations (and `typedef enum` declarations in C-based languages) is the `@const` or `@constant` tag.

Important: The `@const` or `@constant` second-level tag must *not* be the first thing in an enumeration or `typedef` comment. HeaderDoc has a top-level tag with the same name, and cannot readily determine whether you are trying to describe a single constant within the enumeration or have used the wrong top-level tag.

Tag	Example	Identifies	Type
`@constant` `@const`	`@const kSilly A silly return value.`	A constant within an enumeration.	attribute (term & definition) `enum` declarations only

If an enumeration is named in the code, HeaderDoc automatically uses that name (unless you override it with an `@enum` tag). If it is not named, you must supply a name for the enumeration in the HeaderDoc comment. The listings below demonstrate both styles.

Listing 2-23 Example of a named enumeration

```
/*!
     @abstract Categorizes beverages into groups of similar types.
     @constant kSoda Sweet, carbonated, non-alcoholic beverages.
     @constant kBeer Light, grain-based, alcoholic beverages.
     @constant kMilk Dairy beverages.
    @constant kWater Unflavored, non-sweet, non-caloric, non-alcoholic beverages.
     @discussion Extended discussion goes here.
         Lorem ipsum....
*/
enum beverages {
 kSoda = (1 << 6),
 kBeer = (1 << 7),
 kMilk = (1 << 8),
 kWater = (1 << 9)
};
```

Listing 2-24 Example of an anonymous enumeration

```
/*!
     @enum Beverage Categories
     @abstract Categorizes beverages into groups of similar types.
     @constant kSoda Sweet, carbonated, non-alcoholic beverages.
     @constant kBeer Light, grain-based, alcoholic beverages.
     @constant kMilk Dairy beverages.
    @constant kWater Unflavored, non-sweet, non-caloric, non-alcoholic beverages.
     @discussion Extended discussion goes here.
         Lorem ipsum....
*/
enum {
 kSoda = (1 << 6),
 kBeer = (1 << 7),
 kMilk = (1 << 8),
 kWater = (1 << 9)
};
```

Type Definitions

Top-level tag: `@typedef`

The tags that can appear after an `@typedef` tag depend on the contents of the associated declaration. For example, a `typedef enum` declaration can contain anything that an `enum` declaration can contain.

An `@typedef` command can include any of the following:

- `@field` for `typedef struct` declarations
- `@constant` for `typedef enum` declarations
- `@param` for simple `typedef` declarations of individual function pointer types
- `@callback`, `@param`, and `@result` for `typedef struct` declarations containing function pointers as members

Listing 2-25 Typedef for a simple struct

```
/*!
     @typedef TypedefdSimpleStruct
     @abstract Abstract for this API.
     @field firstField Description of first field
     @field secondField Description of second field
     @discussion Discussion that applies to the entire typedef'd simple struct.
         Lorem ipsum....
 */

typedef struct _structTag {
 short firstField;
 unsigned long secondField
} TypedefdSimpleStruct;
```

Listing 2-26 Typedef for an enumeration

```
/*!
     @typedef TypedefdEnum
     @abstract Abstract for this API.
     @constant kCFCompareLessThan Description of first constant.
     @constant kCFCompareEqualTo Description of second constant.
     @constant kCFCompareGreaterThan Description of third constant.
```

```
     @discussion Discussion that applies to the entire typedef'd enum.
         Lorem ipsum....
*/
typedef enum {
 kCFCompareLessThan = -1,
 kCFCompareEqualTo = 0,
 kCFCompareGreaterThan = 1
} TypedefdEnum;
```

Listing 2-27 Typedef for a simple function pointer

```
/*!
     @typedef simpleCallback
     @abstract Abstract for this API.
     @param inFirstParameter Description of the callback's first parameter.
     @param outSecondParameter Description of the callback's second parameter.
     @result Returns what it can when it is possible to do so.
     @discussion Discussion that applies to the entire callback.
         Lorem ipsum...
*/
typedef long (*simpleCallback)(short inFirstParameter, unsigned long long
*outSecondParameter);
```

Listing 2-28 Typedef for a struct containing function pointers

```
/*!
     @typedef TypedefdStructWithCallbacks
     @abstract Abstract for this API.
     @discussion Defines the basic interface for Command DescriptorBlock (CDB)
commands.


     @field firstField Description of first field.


    @callback setPointers Specifies the location of the data buffer. The setPointers
 function has the following parameters:
     @param cmd A pointer to the CDB command interface.
```

```
    @param sgList A pointer to a scatter/gather list.

    @result An IOReturn structure which returns the return value in the structure
 returned.


    @field lastField Description of the struct's last field.
*/
typedef struct _someTag {
 short firstField;
 IOReturn (*setPointers)(void *cmd, IOVirtualRange *sgList);
 unsigned long lastField
} TypedefdStructWithCallbacks;
```

C Preprocessor Macros

Top-level tags: `@define`, `@defined`, `@defineblock`, `@definedblock`, `@/defineblock`, `@/definedblock`

> **Note:** For historical reasons, you can also mark up function-like macros with the `@function` tag. However, this is not recommended.

C preprocessor (`#define`) macros have a few special tags:

Tag	Example	Identifies	Type
`@define` `@defined`	`@define MACRO_NAME`	Legacy top-level tag that provides the macro name.	Term & definition
`@noParse`	`@noParse`	Disables C preprocessor parsing of a macro. The macro will still be included as a `#define` entry in the resulting documentation.	parsing, flag
`@param`	`@param myValue The value to process.`	The name and description of a parameter to a function or callback.	attribute (term & definition) in function-like macros only

Tag	Example	Identifies	Type
`@parseOnly`	`@parseOnly`	Marks macro as "hidden". The macro will be parsed and used by the C preprocessor, but will not be included as a separate `#define` entry in the resulting documentation.	parsing, flag

Listing 2-29 Example of C preprocessor macro

```
/*!
    @defined TRUE
    @abstract Defines the boolean true value.
    @parseOnly
    @discussion Extended discussion goes here.
        Lorem ipsum....
*/
#define TRUE 1
```

Listing 2-30 Example of C preprocessor macro block

```
/*!
    @definedblock Colors of the rainbow
    @abstract Defines some RGB colors.
    @discussion Extended discussion goes here.
        Lorem ipsum....
    @define kInfrared The color infrared.
    @define kRed The color red.
    @define kOrange The color orange.
    @define kYellow The color yellow.
    @define kGreen The color green.
    @define kCyan The color cyan.
    @define kBlue The color blue.
    @define kViolet The color violet.
    @define kUltraviolet The color ultraviolet.
*/
```

```
 #define kInfrared "#000000"
 #define kRed "#FF0000"
 #define kOrange "#FF8000"
 #define kYellow "#FFFF00"
 #define kGreen "#00FF00"
 #define kCyan "#00FFFF"
 #define kBlue "#0000FF"
 #define kViolet "#FF00FF"
 #define kUltraviolet "#000000"

/*! @/definedblock */
```

Declaring Availability Macros

Top-level tag: `@availabilitymacro`

The `@availabilitymacro` tag tells HeaderDoc that whenever the named token appears in a declaration, the token should be deleted and the "Availability:" attribute for that declaration should be set to the string that follows.

Listing 2-31 Example of @availabilitymacro tag

```
/*!
 @availabilitymacro AVAILABLE_IN_MYAPP_1_0_AND_LATER This function is available
in version 1.0 and later of MYAPP.
*/
```

This comment type is usually followed by a #define or similar, but that is not necessary. This HeaderDoc comment is a standalone comment—that is, it does not cause the code after it to be processed in any way. If you want to mark a `#define` as being an availability macro, you should follow this tag with a second HeaderDoc comment for the `#define` itself.

Overriding the Default Data Type: C Pseudoclass Tags

There are three tags provided for C pseudoclasses, such as COM interfaces. The `@class` tag is used for generic pseudoclasses. The `@interface` tag is used for COM interfaces. The `@superclass` tag can be added to an `@class` or `@interface` declaration to modify its behavior.

Table 2-3

Tag	Identifies	Fields
`@superclass`	The name of the superclass.	1

You should mark up any C pseudoclasses in the same way you would mark up a C++ class. Apart from the unusual form of function declarations (in the form of function pointers), the resulting output should be similar to that of a C++ class.

The `@superclass` tag can be used when you have a superclass-like relationship between two C pseudoclasses or COM interfaces. Using this tag will cause the documentation for the specified pseudo-superclass to be injected into the documentation for the current pseudoclass.

The primary purpose for this feature is to reduce the amount of bloat in headers, allowing you to document function pointers in the top-level pseudoclass and then only document the additional function pointers in pseudoclasses that expand upon them.

> **Note:** In order for this feature to work, the super-pseudoclasses must be processed first. If it is in the same header, it must appear before the child pseudoclass. If it is in a separate header, it must appear in a header that the child's header includes, and both headers must be processed at the same time.

Listing 2-32 Example of @class tag

```
/*!
 @class IOFireWireDeviceInterface_t
 @superclass IOFireWireDevice
*/
 typedef struct IOFireWireDeviceInterface_t
{
    IUNKNOWN_C_GUTS;
.
```

```
.
.
}
```

The `@class` tag causes the `typedef struct`that follows the HeaderDoc comment to be treated as a class. This is a frequently-used technique in kernel programming. A slight variation of this tag, `@interface`, is provided for COM interfaces so that they can be identified as such in the TOC. An example of this tag follows:

Listing 2-33 Example of @interface tag

```
/*!
 @interface IOFireWireDeviceInterface_t
 @superclass IOFireWireDevice
*/
 typedef struct IOFireWireDeviceInterface_t
{
    IUNKNOWN_C_GUTS;
.
.
.
}
```

Creating Links Between Symbols

As mentioned in "Second-Level HeaderDoc Tags" (page 53), there are three ways you can use the `@link` tag:

```
@link function_name @/link

or

@link function_name link text goes here @/link

or
```

```
@link //apple_ref/c/func/function_name link text goes here @/link
```

Beginning in HeaderDoc 8.7, you can link to any symbol by its name. (In previous versions of HeaderDoc, you can link to symbols by name if the link target is part of the same `.h` file or in any file processed before it in the same processing run.)

If there are multiple matches for a given name (or if you are using a pre-8.7 version of HeaderDoc and the symbol is parsed afterwards), you may need to explicitly specify which symbol to use. (See "Resolving Conflicting API References" (page 100) to learn about symbol matching precedence.) To avoid the conflict, you include an API reference marker for the symbol instead of its name. For some C++ methods, you may also need to tweak the reference marker so that it does not look like a C-style end-of-comment token. See "Using API References in the @link Tag" (page 99) for details.

Because the `headerdoc2html` script does not know the actual target for these links, it inserts special link request comments into the output. You must then run `gatherheaderdoc` or `resolveLinks` to actually turn those comments into working links.

See "Using resolveLinks to Resolve Cross References" (page 99) to learn more about the resolving process and the various options available to you.

Unknown Tag Handling

To avoid warnings and unexpected output, if you need to use an at sign (@) outside the scope of a HeaderDoc tag, you should quote it by preceding it with a backslash. For example:

```
/*! @header
        For more information, contact myemail\@myaddress.top.
 */
```

If you do not quote the at sign, it will be treated as the start of a tag name, and you may get unexpected behavior.

Beginning in HeaderDoc 8.6 and later, a warning is generated when an unknown tag is encountered, and the tag is converted into text.

Prior to version 8.6, unknown tags were partially removed. The initial at sign (@) was deleted, leaving only the content following it.

Adding Arbitrary Attributes

@@@ WRITE ME @@@

`@attribute`

`@attributelist`

`@attributeblock`

Basic HeaderDoc Configuration

You can set values for some commonly altered variables that affect HeaderDoc's behavior and output style. This chapter describes the configuration file format, location, and the available configuration file keys.

Configuration File Format

A variable can be assigned a value in any of these places, but only the last value read for a given variable will affect the output of a run of the script. If you are happy with the default values for these variables (as described above), you don't need to provide a configuration file. If you want to change just one or more values, provide a configuration file that declares just those values.

The format of the configuration file is this:

```
key1 => value1
key2 => value2
```

HeaderDoc looks for these variables in three places, in this order:

1. In the script itself (see the declaration of the `%config` hash near the top of `headerdoc2html` or `headerDoc2HTML.pl`).
2. For HeaderDoc 8.9 and later, in `/usr/share/headerdoc/conf` (open source builds) or `/path/to/Xcode.app/Contents/Developer/usr/share/headerdoc/conf` (when installed as part of the developer tools).
3. In the main Library folder, in `/Library/Preferences/com.apple.headerDoc2HTML.config` (in most versions)
4. In the home directory of the user, in `$HOME/Library/Preferences/com.apple.headerDoc2HTML.config`
5. In a file named `headerDoc2HTML.config` in the same folder as the script.

In iterating through these locations, HeaderDoc retains the last value that it sees. Thus, the most local copy of any variable overrides any more generic value.

Configuration File Keys

Currently, the HeaderDoc configuration file lets you set the following things:

- General tool behavior—described in "Behavioral Settings Keys" (page 55).
- Output format—described in "General Output Style Settings" (page 56).
- CSS stylesheet fragments and references to external style sheets to insert into the output—described in "General CSS Keys" (page 58).
- CSS stylesheet fragments for specific parts of declarations—described in "Declaration CSS Keys" (page 59).

Behavioral Settings Keys

This section describes configuration keys that control HeaderDoc's general behavior, not including any output formatting or styling.

apiUIDPrefix

The prefix for named anchors (by default, `apple_ref`). In the output, HeaderDoc adds a self-describing named anchor near each API declaration—for example `<a name="//apple_ref/c/func/CFArrayAppendValue">`. These can be useful for index generation and other purposes. See "Symbol Markers for HTML-Based Documentation" (page 92) for more information.

compositePageName

The name of the file containing the printable HTML page (by default, `CompositePage.html`). Not used if `classAsComposite` is 1.

defaultFrameName

The name of the file containing the frameset instructions (by default, `index.html`).

externalAPIUIDPrefixes

A space-separated list of prefixes for API references. When `gatherheaderdoc` runs `resolveLinks`, it passes this list of prefixes to `resolveLinks`. This allows you to use (multiple) API reference prefixes other than apple_ref.

For more information, see "Symbol Markers for HTML-Based Documentation" (page 92).

externalXRefFiles

A space-separated list of paths to external files, each of which contains a list of cross references outside the current document. When `gatherheaderdoc` runs `resolveLinks` to link together cross-referenced

content, it passes these external cross-reference files to `resolveLinks` so that you can look up API references (apple_ref-style markup) in other documents.

Note: Generally, if you are using external cross-reference files, you should be running `resolveLinks` manually rather than using this setting.

For more information, see "Symbol Markers for HTML-Based Documentation" (page 92).

ignorePrefixes

A list of tokens to leave out of the final output if they occur at the start of a line (before any other non-whitespace characters). Although this feature still exists, it is usually better to use C preprocessor directives.

masterTOCName

The name of the file containing the master table of contents for a series of headers (by default, `masterTOC.html`). (This variable is used by the `gatherheaderdoc` script, and can be overridden on the command line.)

IDLLanguage

IDL files normally produce `apple_ref` markers with the language `"idl"`. However, an IDL file is inherently language-neutral. This flag allows you to tell HeaderDoc to use a different language in `apple_ref` markers resulting from processing an IDL file.

Legal values are, in practice, any arbitrary URL-encoded string, but ideally should be valid programming languages as defined in the `apple_ref` specification. See "Symbol Markers for HTML-Based Documentation" (page 92) for the specification.

General Output Style Settings

This section describes overall output style (not including CSS style sheets, which are described in "General CSS Keys" (page 58) and "Declaration CSS Keys" (page 59)).

appleTOC

Specifies the Apple TOC format. This format requires extensive JavaScript and CSS support, and thus is not very useful outside of the developer.apple.com website. It is documented only for completeness.

classAsComposite

By default, HeaderDoc splits the documentation on the right side into several files. This setting causes classes to be output in a single composite page instead.

copyrightOwner

The copyright notice that appears at the bottom of the HTML pages. Unless you specify a value, no copyright will appear.

dateFormat

A string specifying the date format to be used by HeaderDoc. This date format is specified using standard time formatting flags. For examples of valid date formats, see the man page for `strftime`.

groupHierLimit

The maximum number of entries allowed in a list of headers, functions, and so on before `gatherheaderdoc` inserts jump links to particular letters within the list. If this key is absent, no letter links are inserted. If you set this key, you *must* also set the `groupHierSubgroupLimit` to a positive integer value.

groupHierSubgroupLimit

The maximum number of entries that should ideally appear in a single letter grouping. When this limit is exceeded, a new group begins as soon as an entry is reached whose first two letters differ from those of the current entry. This key is mandatory if `groupHierLimit` is set, and must be a positive integer.

htmlFooter

A string (generally a server-side include directive) that HeaderDoc will insert into the bottom of each right-side and composite HTML page if you specify the `–H` flag on the command line. For longer headers, use `htmlFooterFile`.

htmlFooterFile

A file containing a longer HTML footer. The contents of this file will be added to the end of each content page if you specify the `–H` flag on the command line.

htmlHeader

A string (generally a server-side include directive) that HeaderDoc will insert into the top of each right-side and composite HTML page if you specify the `–H` flag on the command line. For longer headers, use `htmlHeaderFile`.

htmlHeaderFile

A file containing a longer HTML header. The contents of this file will be added at the top of each content page if you specify the `–H` flag on the command line.

stripDotH

This option causes `gatherheaderdoc` to strip the trailing `.h` from the names of header filenames in header lists.

TOCFormat

Chooses the TOC format style for individual documents. Legal values are `default` (new style with disclosure triangles), `frames` (old style), or `iframes` (8.7 style).

This option replaces the `–F` flag, though that flag is still supported for now.

TOCTemplateFile

Specifies a TOC template file to use instead of the built-in TOC template. For more information, see "Creating a TOC Template File" (page 62).

TOCTemplateEncoding

The encoding used by your TOC template file. The `gatherHeaderDoc` tool uses this to ensure that any date stamps inserted into master TOCs are in the correct encoding.

useBreadcrumbs

Setting this option to `1` tells HeaderDoc that you intend to use an external tool to create breadcrumb links in your documents. When you specify this option, it disables the insertion of the "[Top]" link in the table of contents, since it is not necessary if you have such a tool. Because such breadcrumbs are site-specific, no such tools are provided as part of HeaderDoc.

General CSS Keys

externalStyleSheets

A space-separated list of paths to external style sheet files on the server or destination volume. For example, if you set externalStyleSheets to /CSS/mysheet.css, HeaderDoc will insert the following:

```
<link rel="stylesheet" type="text/css" href="/CSS/mysheet.css">
```

These style sheets are inserted prior to any HeaderDoc-generated styles.

Note: Using this option disables the built-in HeaderDoc styles. For your convenience, these built-in styles are listed in "Built-in HeaderDoc Styles" (page 61).

---

externalTOCStyleSheets

Like externalStyleSheets, this is a space-separated list of paths to external style sheet files on the server or destination volume. If no TOC style sheets are specified, the style sheets specified in `externalStyleSheets` will be used.

Note: Using this option disables the built-in HeaderDoc styles. For your convenience, these built-in styles are listed in "Built-in HeaderDoc Styles" (page 61).

---

styleImports

A string of CSS to be inserted just prior to HeaderDoc-generated CSS, but after any external style sheets. This was originally intended to support the `@import` directive to import an external style sheet, but may be used for any arbitrary CSS content.

Note: Using this option disables the built-in HeaderDoc styles. For your convenience, these built-in styles are listed in "Built-in HeaderDoc Styles" (page 61).

styleSheetExtrasFile

A file containing local HeaderDoc-specific CSS. The contents of the file specified will be inserted at the end of the built-in HeaderDoc styles (after any styles specified by the HeaderDoc declaration styles, such as `varStyle`).

Note: This option is the *only* style sheet option that does *not* disable the built-in HeaderDoc styles.

tocStyleImports

Similar to `styleImports`, this is a string of CSS to be inserted just prior to HeaderDoc-generated CSS, but after any external style sheets. This was originally intended to support the `@import` directive to import an external style sheet, but may be used for any CSS content.

If no TOC style imports are specified, the value of `styleImports` will be used for the TOC.

Note: Using this option disables the built-in HeaderDoc styles. For your convenience, these built-in styles are listed in "Built-in HeaderDoc Styles" (page 61).

Declaration CSS Keys

These contain CSS formatting for various parts of declarations. For example:

```
funcNameStyle => background:#ffffff; color:#000000;
```

sets function names to be displayed in black text on a white background.

charStyle

style for characters ('a')

commentStyle

style for comments

funcNameStyle

style for function names

keywordStyle

style for keywords

numberStyle
style for numbers

paramStyle
style for function parameters

preprocessorStyle
style for preprocessor directives

stringStyle
style for strings

textStyle
style for normal text if declarations (mainly parentheses, punctuation, and spaces)

typeStyle
style for data types

varStyle
style for variable names

Configuration File Example

Listing 3-1 (page 60) is an example of a very basic HeaderDoc configuration file. Several additional examples are included as part of the HeaderDoc distribution.

Listing 3-1 Sample HeaderDoc configuration file

```
copyrightOwner => My Great Software Company
defaultFrameName => default.html
compositePageName => PrintablePage.html
masterTOCName => TOCCentral.html
apiUIDPrefix => greatSoftware
ignorePrefixes=> CF_EXTERN|CG_EXTERN
htmlHeader=>
dateFormat=> %m/%d/%Y
```

Built-in HeaderDoc Styles

Many of the CSS options in HeaderDoc disable the built-in styles so that it is easier to override those styles in external style sheets. The built-in styles are listed below for your convenience.

Listing 3-2 Built-in HeaderDoc CSS Styles

```
a:link {text-decoration: none; font-family: lucida grande, geneva, helvetica,
arial, sans-serif; font-size: small; color: #0000ff;}

a:visited {text-decoration: none; font-family: lucida grande, geneva, helvetica,
arial, sans-serif; font-size: small; color: #0000ff;}

a:visited:hover {text-decoration: underline; font-family: lucida grande, geneva,
helvetica, arial, sans-serif; font-size: small; color: #ff6600;}

a:active {text-decoration: none; font-family: lucida grande, geneva, helvetica,
arial, sans-serif; font-size: small; color: #ff6600;}

a:hover {text-decoration: underline; font-family: lucida grande, geneva, helvetica,
 arial, sans-serif; font-size: small; color: #ff6600;}

h4 {text-decoration: none; font-family: lucida grande, geneva, helvetica, arial,
sans-serif; font-size: tiny; font-weight: bold;}

body {text-decoration: none; font-family: lucida grande, geneva, helvetica, arial,
 sans-serif; font-size: 10pt;}
```

Advanced HeaderDoc Configuration and Features

HeaderDoc contains a number of advanced features intended for users with more complex needs. This chapter describes some of these features.

Creating a TOC Template File

TOC template files are basically ordinary HTML files. They can contain any HTML content. In addition to HTML content, they can also contain conditional HTML content—that is, content that is only included if certain conditions are met. Finally, they can include various lists.

The template support is particularly powerful when combined with support for frameworks (which, for HeaderDoc purposes, is essentially a loose grouping of related documentation stored in the same output directory).

Here are the special tags that indicate conditional or list content:

`$$title@@`
: Inserts "Foo Documentation" where Foo is the framework name.

`$$tocname@@`
: Inserts the name of the main TOC file. Useful when used with multiple landing page templates, as described in "Using Multiple Landing Page Templates" (page 66).

`$$framework@@`
: Inserts the full framework name, as specified by the `@framework` tag in the `.hdoc` file.

`$$frameworkabstract@@`
: Inserts the framework abstract, as specified by the `@abstract` tag in the `.hdoc` file.

`$$frameworkdir@@`
: Inserts the framework's "short name". This is determined by taking the filename of the "`.hdoc`" file and stripping off the `.hdoc` extension). This name is also prepended to the name of additional landing page template, as described in "Using Multiple Landing Page Templates" (page 66).

`$$frameworkdiscussion@@`
: Inserts the framework discussion, as specified by the `@discussion` tag (or implicitly as part of the `@framework` tag) in the `.hdoc` file.

`$$frameworkuid@@`

Inserts a framework UID anchor.

`$$headersection@@`

Start of conditional block for headers. If there are no headers listed, content between this tag and the closing conditional block tag will not appear.

`$$/headersection@@`

End of conditional block for headers.

`$$headerlist@@`

A list of all headers in the output directory.

`$$classsection@@`

Start of conditional block for classes. If there are no classes listed, content between this tag and the closing conditional block tag will not appear.

`$$/classsection@@`

End of conditional block for classes.

`$$classlist@@`

A list of all classes in the output directory.

`$$categorysection@@`

Start of conditional block for categories. If there are no categories listed, content between this tag and the closing conditional block tag will not appear.

`$$/categorysection@@`

End of conditional block for categories.

`$$categorylist@@`

A list of all categories in the output directory.

`$$protocolsection@@`

Start of conditional block for protocols. If there are no protocols listed, content between this tag and the closing conditional block tag will not appear.

`$$/protocolsection@@`

End of conditional block for protocols.

`$$protocollist@@`

A list of all protocols in the output directory.

`$$datasection@@`

Start of conditional block for data (globals and constants). If there are no data elements listed, content between this tag and the closing conditional block tag will not appear.

`$$/datasection@@`

End of conditional block for data (globals and constants).

`$$datalist@@`
: A list of all data elements in the output directory.

`$$typesection@@`
: Start of conditional block for types. If there are no types listed, content between this tag and the closing conditional block tag will not appear.

`$$/typesection@@`
: End of conditional block for types.

`$$typelist@@`
: A list of all types in the output directory.

`$$functionsection@@`
: Start of conditional block for functions or methods. If there are no functions or methods listed, content between this tag and the closing conditional block tag will not appear.

`$$/functionsection@@`
: End of conditional block for functions or methods.

`$$functionlist@@`
: A list of all functions/methods in the output directory.

List tags default to a raw list (single column) with no border. However, you can change the number of columns, the table width, and border quite easily. For example:

```
    $$functionlist cols=3 order=down atts=border="0" cellpadding="1" cellspacing="0"
 width="420"@@
```

specifies that the table will be three columns, listed down the first column, then down the next column, and so on. It also specifies that the additional attributes `border`, `cellpadding`, `cellspacing`, and `width` will be inserted into the table tag automatically. Note that the `atts` parameter must be the last parameter listed.

Warning: The order of the arguments to the list commands is important. The order of options is listed below

`nogroups`
: The `gatherheaderdoc` tool normally separates entries by TOC grouping. If you want this list to include everything in a single list, add this flag. For an example, see the alphabetical list of all OS X manual pages as part of *OS X Man Pages* .

`cols`

Specifies the number of columns in the table. (Note that the number of rows cannot be specified, as it is calculated based on the number of columns and the number of entries in the table.) For example, you might specify `cols=3`.

`order`

Specifies whether the table should read across or down. If you specify `order=across`, the first entry will be in the upper left cell, the second one will be to the right, and so on. If you specify `order=down`, the second entry will be below the first entry. The default is down.

`trclass`

Specifies a CSS class to be applied to the `<tr>` (table row) tags within the table. For example, you might specify `trclass=toctrclass`.

`tdclass`

Specifies a CSS class to be applied to the `<td>` (table data cell) tags within the table. For example, you might specify `tdclass=toctdclass`.

`notable`

Disables generation of tables. If you specify this option, each entry will be separated by a `<br>` (line break) tag followed by a newline. This is primarily intended for generating a list that can be easily processed with custom tools, but it may be combined with CSS to create some interesting and useful layouts as well.

`addempty`

This option tells `gatherheaderdoc` to include blank cells containing a non-breaking space to fill in unused slots in the last line of the table. The default, `addempty=0`, will simply close the final line of the table early. To add extra empty cells (as needed) to fill the last line in the table, specify `addempty=1`.

This usually matters very little unless you have table borders turned on (`atts=border=1`, for example).

`atts`

Specifies a list of attributes to be added to the `<table>` tag. These are not CSS attributes, though you could specify CSS attributes by specifying `atts=style="CSS props here"`. Everything up to the closing `@@` marker is included as part of the `atts` option.

For example:

```
    $$functionlist nogroups cols=3 order=down trclass=mytrclass tdclass=mytdclass
 notable addempty=1 atts=border="0" cellpadding="1" cellspacing="0" width="420"@@
```

Using Multiple Landing Page Templates

HeaderDoc is not limited to a single landing page template. You can generate multiple landing pages with different content if desired. To do this, you might create two template files called `toctemplate.html` and `functions.tmpl`, then add a line in your configuration file like this:

```
TOCTemplateFile => toctemplate.html functions.tmpl
```

When you run `gatherheaderdoc`, you will now get two HTML landing pages, one for each template.

The first template file, `toctemplate.html`, is treated as the "main" template page. The `gatherheaderdoc` tool will generate a landing page based on that template with the filename specified by the `masterTOCName` variable in the configuration file (`masterTOC.html` by default).

After the first template file, each additional template file (`functions.tmpl`, in this case) is used to produce an HTML landing page whose name is derived from the framework's "short name" (the name of the `.hdoc` file with the `.hdoc` extension stripped off the end), followed by a dash, followed by the template filename (without any "`.html`" or "`.tmpl`" extensions), followed by "`.html`".

For example, if the `.hdoc` file is called `MyFramework.hdoc`, this second index file would be called `MyFramework-functions.html`.

Since these templates can be used for generating multiple documents, you should not specify this entire path in your template files, however. Instead, you should specify it relative to the framework name. To do this, in your `toctemplate.html` file, you should link to the functions index like this:

```
<A href="$$frameworkdir@@-functions.html">Functions Index</A><p>
```

The framework's "short name" will automatically be substituted in place of the `$$frameworkdir@@` keyword. Similarly, in the functions template, you can link to the main TOC like this:

```
<A href="$$tocname@@">Headers Index</A><p>
```

This will ensure that your template will generate valid links even if you change the name of the MasterTOC in your configuration file.

Example gatherheaderdoc Template

The following is an example template for `gatherheaderdoc`:

```
<html>
 <head>
 <title>API Reference: Device Drivers (Kernel/IOKit)</title>
 <style type="text/css"><!--#pagehead {
    FONT-WEIGHT: bold; FONT-SIZE: 32px; COLOR: #000000;
    FONT-FAMILY: lucida grande, geneva, helvetica, arial, sans-serif; }
    td { font-size: 10px; } a:link {text-decoration: none;
    font-family: lucida grande, geneva, helvetica, arial, sans-serif;
    color: #0000ff;} a:visited {text-decoration: none;
    font-family: lucida grande, geneva, helvetica, arial, sans-serif;
    color: #0000ff;} a:visited:hover {text-decoration: underline;
    font-family: lucida grande, geneva, helvetica, arial, sans-serif;
    color: #ff6600;} a:active {text-decoration: none;
    font-family: lucida grande, geneva, helvetica, arial, sans-serif;
    color: #ff6600;} a:hover {text-decoration: underline;
    font-family: lucida grande, geneva, helvetica, arial, sans-serif;
    color: #ff6600;} h4 {text-decoration: none;
    font-family: lucida grande, geneva, helvetica, arial, sans-serif;
    font-size: tiny; font-weight: bold;} body {text-decoration: none;
    font-family: lucida grande, geneva, helvetica, arial, sans-serif;
    font-size: 10pt;} -->
 </style>
 </head>
 <Meta name="ROBOTS" content="NOINDEX">

<body bgcolor="#ffffff">
<center>

<!-- start of header -->
<!--#include virtual="/path/to/header.html"-->
<!-- end of header -->

<table border="0" cellpadding="0" cellspacing="0" width="600">

    <tr height="5">
```

```
    <td width="600" height="5"><br>
    </td>
</tr>
<tr>
    <td width="600">
        <div id="pagehead">$$framework@@</div>
    </td>
</tr>
<tr height="10">
    <td width="600" height="10"><br>
    </td>
</tr>
<tr>
    <td valign="top" width="600"><font face="Geneva,Helvetica,Arial"
    size="2"><span id="bodytext"> $$frameworkdiscussion@@ </span></font>
    </td>
</tr>
<tr height="10">
    <td height="10" width="600"></td>
</tr>
<tr height="5">
    <td height="5" width="600">
        <hr alt="">
        <br>
    </td>
</tr>
<tr>
    <td width="600" align="center" valign="top">
        <H2>Headers</H2>

            $$headerlist cols=3 order=down atts=border="0"
            cellpadding="1" cellspacing="0" width="420"@@
    <H2>Functions</H2>
            $$functionlist cols=3 order=down atts=border="0"
```

```
                cellpadding="1" cellspacing="0" width="420"@@
        </td>
    </tr>
</table>
</center>
</body>
</html>
```

Using the C Preprocessor

Beginning in HeaderDoc 8.5, HeaderDoc contains a basic C preprocessor implementation (enabled with the `-p` flag). Because HeaderDoc does not have access to the full compile-time environment of the headers, its behavior may differ from normal C preprocessors in certain cases. This section describes some of those differences.

Parsing Rules

Most `#define` macros are not parsed by default, even if the preprocessor is enabled. This permits you as the user to choose which macros to process.

Macros are processed if any of the following are true:

- They are preceded by a HeaderDoc comment block.
- They appear between the beginning and end of a class that is preceded by a HeaderDoc comment block.

The reason for this second case is a side-effect of the way that HeaderDoc parses classes to ensure that lines are processed in the order in which they appear in the file (which is necessary for a preprocessor to even be possible). For maximum control, preprocessor directives should be at the start of the file, outside of class braces.

Multiply-Defined Macros

HeaderDoc does not attempt to handle `#if`, `#ifdef`, or `#ifndef` directives. This may, in certain circumstances, result in multiple definitions of a `#define` directive if the preprocessor is enabled. As with most preprocessors, all such definitions are ignored except for the one that appears first in the file.

This is made slightly more complicated by the parsing rules described in "Parsing Rules" (page 69).

Embedded HeaderDoc Comments in a Macro

With most data types, HeaderDoc comments appearing inside the data type are associated with the data type itself. This is normally true for `#define` macros as well. However, that behavior would create a problem when the C preprocessor is enabled, as it is reasonable to allow macros to define contents to be blown into a class, and those contents could potentially include HeaderDoc markup.

For this reason, when the C preprocessor is enabled, embedded HeaderDoc processing is disabled for `#define` macros. Any HeaderDoc markup within the body of such a macro will be blown in wherever the macro is used, and will only be processed in the resulting context.

While HeaderDoc does allow a macro to insert multiple declarations and HeaderDoc comment blocks within a class, it does not allow this outside of a class. When a macro inserts contents outside of a class scope, parsing will end at the end of the first declaration and any other contents inserted by the macro will be skipped.

Handling of #include

HeaderDoc's implementation of `#include` behaves differently than you might expect. The differences include the following:

- No notion of paths.

  Because the include paths are not specified as they are with a compiler, HeaderDoc cannot reasonably determine that `<dir1/file.h>` and `<dir2/file.h>` are distinct. For this reason, processing files with the same name in different directories is discouraged.

- Mandatory recursion protection.

  Because HeaderDoc does not process `#if`, `#ifdef`, and `#ifndef` conditionals, HeaderDoc enforces recursion protection by not allowing a file to get processed twice. Once a file is processed, a precompiled copy of its macros is stored for future use, and is automatically inserted whenever another `#include` requests it.

  This causes two side-effects. First, a `#include` cannot be altered in a context-dependent way—that is, if a header incudes `<a.h>` and then `<b.h>`, the macros defined in `<a.h>` will not affect the parse of `<b.h>` unless `<b.h>` includes `<a.h>` on its own.

  Second, `#include` behaves much like `#import`. The result is that if `<a.h>` includes `<b.h>` which includes `<c.h>` which includes `<a.h>`, the definitions leading up to the re-inclusion of `<a.h>` will not affect the way `<a.h>` is parsed.

- Macro contents will be shown in the documentation output.

Rather than try to carry around some notion of the original tokens read from the file, HeaderDoc inserts macros into the parse tree as if the modified version had been read from the file. This means that it is not possible, for example, for HeaderDoc to show you the unaltered definition of a macro that includes another macro.

These differences generally do not affect headers written in a typical fashion, but may cause problems if you are using preprocessor directives in a nonstandard way.

Other Issues

A few common function-like preprocessor macros are predefined within HeaderDoc itself to avoid parse problems with I/O Kit headers. These will probably not affect you, but you should be aware of them.

Because HeaderDoc does not strip comments prior to processing macros (since doing so would remove HeaderDoc markup), the preprocessor may behave in subtly different ways. In particular, newlines are preserved, and any closing single-line (`//`) comments will automatically be converted into a multi-line (`/* */`) comment to avoid causing the rest of the line to disappear when that macro actually gets used.

Finally, HeaderDoc does do basic string and character handling, even within macros. As a result, mismatched single and double quotes within a `#define` macro may cause serious problems.

What if I Don't Want to See the Macros in the Documentation?

Most of the time, having `#define` macros defined in the documentation is helpful. In some cases, though, the macros get so big and ugly that you just want to get rid of them. For this reason, HeaderDoc has the `@parseOnly` tag.

For example:

```
/*! This is an ugly internal macro. @parseOnly */
#define CreateStructors \
    /*! Constructor */ \
    blah(); \
    /*! Destructor */ \
    ~blah();
```

By adding this tag at the end of the HeaderDoc comment block for the macro, the macro will be parsed and used by the preprocessor, but will not appear in the documentation.

Using the MPGL Suite

In addition to the main `headerdoc2html` and `gatherheaderdoc` scripts, the HeaderDoc suite contains additional utilities for generating manual pages (using the mdoc macro set).

The Man Page Generation Language (MPGL) suite contains two utilities: `xml2man` and `hdxml2manxml`. The `xml2man` utility converts an mdoc-like XML dialect, the Man Page Generation Language (MPGL) into manual pages. The `hdxml2manxml` utility converts HeaderDoc XML output into a series of files that can then be processed using `xml2man`.

Both commands have a very simple syntax. Neither takes any arguments.

```
hdxml2manxml filename1 filename2 ... filenameN
xml2man inputfile.mxml [ outputfile.1 ]
```

In the case of `xml2man`, the output filename is generally left blank.

The remainder of this chapter describes the XML dialect used by these utilities.

Man Page Generation Language (MPGL) Dialect

This section describes the basic syntax of the Man Page Generation Language (MPGL). Portions of the syntax are abridged due to complexity. For information on these details, see the examples later in this chapter.

> **Note:** Many versions of `man` are exceptionally picky about blank lines. While the `xml2man` translator attempts to remove most of these, you should still avoid leaving blank lines in the input files.

The MPGL syntax includes a subset of mdoc. All text is unjustified, and some redundancy was reduced. In particular, the `usage` section in an MPGL file provides the source information for both the SYNOPSIS and OPTIONS sections of a traditional man page. Beyond those changes, if you are familiar with the mdoc macro set, you should feel right at home.

At the top level (within the outer `<manpage>` tag), an MPGL page consists of some or all of the following large blocks:

Table 5-1 MPGL block tags

Block tag	Description
`<docdate>`	The last modified date of the manual page.
`<description>`	A description of the technology as a whole. This is the first major section of the resulting manual page.
`<doctitle>`	The title of the manual page.
`<os>`	The operating system for which the manual page was written.
`<section>`	The man section in which the manual pages should appear.
`<names>`	Names and descriptions of functions or tools described in this manual page (see example for syntax).
`<usage>`	Command-line usage or function parameters (see example for syntax).
`<returnvalues>`	Function return value (text description).
`<environment>`	Interaction with environment variables.
`<files>`	Files used by a command-line tool.
`<examples>`	Usage examples.
`<diagnostics>`	Troubleshooting information.
`<errors>`	Function error values (generally restricted to those returned via the `errno` global variable).
`<seealso>`	Cross-references to other manual pages (see example).
`<conformingto>`	Standards to which a tool or function conforms.
`<history>`	Historical information.
`<bugs>`	Known bugs in a tool or function.

Any field can contain either a block of raw text or the following subset of XHTML:

Table 5-2 XHTML tags supported by MPGL

XHTML tag	Description
`<p>`	paragraph

XHTML tag	Description
<blockquote>	indented block
<tt>	indented literal text or code
	unordered (bullet) list
	ordered (numbered) list
	list item (within a list)
<code>	literal text
<dl>	term and definition list
<dt>	term (within a term and definition list)
<dd>	definition (within a term and definition list)

Any field can also contain any of the following MPGL-specific inline tags:

Table 5-3 Additional MPGL-specific inline tags

Tag	Description
<path>	path name
<function>	function name
<command>	command name
<manpage>	man page cross-reference (see example)

A Simple Function Example

Listing 5-1 (page 74) is an example of how to write an MPGL manual page for a function.

Listing 5-1 A simple MPGL example for a function

```
<manpage>
<docdate>August 28, 2002</docdate>
<doctitle>Document title</doctitle>
```

```
<os>OS X</os>
<section>3</section>
<names>
        <name>foo<desc>This is foo's description</desc></name>
        <name>bar<desc>This is bar's description</desc></name>
</names>

<usage>
        <func><type>int</type><name>foo</name>
            <arg>int k<desc>This is a k.</desc></arg>
            <arg>char *b<desc>This is a b.</desc></arg>
        </func>
</usage>

<returnvalues>
        <p>Returns kIONotANumber if you can't count.</p>
        <p>Returns kIOMoron this if you REALLY can't count.</p>
</returnvalues>

<environment>
        TEXT
</environment>

<files>
        <file>/path/to/filename<desc>This is a waste of time</desc></file>
        <file>/path/to/another/filename<desc>This is also a waste of
time</desc></file>
</files>

<examples>
        TEXT
</examples>

<diagnostics>
        TEXT
```

```
</diagnostics>

<errors>
        TEXT
</errors>

<seealso>
        <p>This is a text container, really, but generally contains
        lines like this:</p>
        <manpage>foo<section>1</section>, </manpage>
        <manpage>bar<section>3</section></manpage>
</seealso>

<conformingto>
        <p>Here's a list of conformance:</p>
        <ul>
            <li>Single UNIX Specification</li>
            <li>POSIX</li>
        </ul>
</conformingto>

<history>
        TEXT
</history>

<bugs>
        <p>Here are some bugs:</p>
        <p>
        <ol>
                <li>Bug one....</li>
                <li>Bug two....</li>
                <li>Bug three....</li>
        </ol>
        </p>
```

```
        <p>I think that pretty much covers it.</p>
</bugs>
</manpage>
```

A Simple Command Example

Listing 5-2 (page 77) is an example of how to write an MPGL manual page for a single command or a series of commands with the same syntax.

Listing 5-2 A simple MPGL example for a command

```
<manpage>
<docdate>August 28, 2002</docdate>
<doctitle>Document title</doctitle>
<os>Darwin</os>
<section>1</section>
<names>
        <name>foo<desc>this is a description</desc></name>
        <name>bar<desc>this is also a description</desc></name>
</names>

<usage>
        <flag optional="1">a<arg>attributes</arg><desc>This is the atts
flag</desc></flag>
        <flag>d<arg>date</arg><desc>This is the date flag</desc></flag>
        <flag>x<desc>This is the -x flag</desc></flag>
        <arg>filename<desc>This is the filename</desc></arg>
</usage>

<returnvalues>
        <p>Returns kIONotANumber if you can't count.</p>
        <p>Returns kIOMoron if you REALLY can't count.</p>
</returnvalues>

<environment>
```

```
        TEXT
</environment>

<files>
        <file>/path/to/filename<desc>This is a waste of time</desc></file>
        <file>/path/to/another/filename<desc>This is also a waste of
time</desc></file>
</files>

<examples>
        TEXT
</examples>

<diagnostics>
        TEXT
</diagnostics>

<errors>
        TEXT
</errors>

<seealso>
        <p>This is a text container, really, but generally contains
        lines like this:</p>
        <manpage>foo<section>1</section>, </manpage>
        <manpage>bar<section>3</section></manpage>
</seealso>

<conformingto>
        <p>Here's a list of conformance:</p>
        <ul>
            <li>Single UNIX Specification</li>
            <li>POSIX</li>
        </ul>
```

```
        <p>Here's a definition list:</p>
        <dl>
            <dd>foo_aaa</dd>
                <dt>This is foo</dt>
            <dd>bar</dd>
                <dt>This is bar</dt>
        </dl>

</conformingto>

<history>
        This program should be history....
</history>

<bugs>
        <p>Here are some bugs:</p>
        <p>
        <ol>
                <li>Bug one....</li>
                <li>Bug two....</li>
                <li>Bug three....</li>
        </ol>
        </p>
        <p>I think that pretty much covers it.</p>
</bugs>
</manpage>
```

A Multi-Command Example

Listing 5-3 (page 79) is an example of how to write an MPGL manual page for multiple commands in a single page.

Listing 5-3 An MPGL example for multiple commands

```
<manpage>
```

```
<docdate>August 28, 2002</docdate>
<doctitle>Document title</doctitle>
<os>Darwin</os>
<section>1</section>
<names>
        <name>hdxml2manxml<desc>HeaderDoc XML to MPGL translator</desc></name>
        <name>xml2man<desc>MPGL to mdoc (man page) translator</desc></name>
        <name>examplemc<desc>MPGL to mdoc (man page) translator</desc></name>
</names>

<usage>
        <command name="hdxml2manxml">
                <arg>filename [ filename ... ]<desc>the filename(s) to be
processed</desc></arg>
        </command>
        <command name="xml2man">
                <arg>filename<desc>This is the filename</desc></arg>
                <arg optional="1">output_filename<desc>This is the
filename</desc></arg>
        </command>
        <command name="example">
                <arg>filename<desc>This is the filename</desc></arg>
                <arg optional="1">output_filename<desc>This is the
filename</desc></arg>
        </command>
        <command name="example">
                <arg>filename [ filename ... ]<desc>the filename(s) to be
processed</desc></arg>
                <flag optional="1">c<arg>time_to</arg><arg
optional="1">crash</arg><desc>Seems like a useful flag</desc></flag>
        </command>
</usage>

<environment>
        <p>The <name>xml2man</name> program was designed to convert Man Page
        Generation Language (MPGL) XML files into mdoc-based manual pages.
```

```
        The MPGL is a fairly direct translation of mdoc to XML.</p>

        <p>The <name>hdxml2manxml</name> tool was designed to translate
        from headerdoc's XML output to an mxml file for use with xml2man.</p>
</environment>

<seealso>
        <p>For more information on xml2man, see</p>
        <manpage>xml2man<section>1</section>, </manpage>
        <manpage>hdxml2manxml<section>1</section>, </manpage>
</seealso>

</manpage>
```

Testing HeaderDoc

Beginning in version 8.7, HeaderDoc includes a regression test suite that makes it easier to modify the parser code without causing regressions. This chapter describes how to use the test suite.

Obtaining a HeaderDoc Tarball

Beginning in HeaderDoc 8.8, the test suite is installed as part of the HeaderDoc installation. Thus, you can run the test suite without downloading a source tarball. However, the main reason to run it is if you are modifying the source code, in which case it is much easier to work with a standalone copy of HeaderDoc.

You can download the HeaderDoc tarball from www.opensource.apple.com. Click the latest version of OS X on that page, then search for `headerdoc` on the resulting page and click the download link to the right.

Running the Tests

You can run tests in two ways: all at once or individually.

- To run all of the tests at once, type the following commands:

```
# From an installed copy:
headerdoc2html -T run

# From a standalone copy:
cd /path/to/headerdoc-version
./headerDoc2HTML.pl -T run
```

  HeaderDoc runs the complete battery of tests and produces a summary of the results at the end.

- To run a single test or a group of tests, specify their path or paths after the `run` subcommand. For example:

```
# From an installed copy
headerdoc2html -T run
/path/to/Xcode.app/Contents/Developer/usr/share/headerdoc/testsuite/parser_tests/PHP_function_1.test
```

```
# From a standalone copy:
./headerDoc2HTML.pl -T run testsuite/parser_tests/PHP_function_1.test
```

Handling Test Failures

When using the `run` subcommand, you get only a summary output telling what test failed. Finding out exactly what went wrong can be more challenging. Here are the basic steps:

1. Use the `update` subcommand as though you were trying to update test results (`headerDoc2HTML.pl -T update` *[optional list of tests]*). When a test fails, HeaderDoc displays the test result differences.
2. Type `less` and press return to get a contextual diff of some of the more verbose sections of the result.
3. If you see that the change is expected (if you made a change to the code and the new results reflect a desired change), type `confirm` to update the test results with the new data.

   Important: This step will fail if you are running an installed copy of HeaderDoc because the test suite is not installed in a writable location. See "Obtaining a HeaderDoc Tarball" (page 82) to learn how to download and extract a HeaderDoc source tarball.

4. If the test results are not as expected, type `skip` to go on to the next test without updating the test results.

Creating a Test

Important: Before you can create a test, you must download and extract a HeaderDoc source tarball. You cannot create a test in the installed copy of HeaderDoc because the test suite is not installed in a writable location. See "Obtaining a HeaderDoc Tarball" (page 82) to learn how to do this.

Creating a test is somewhat more involved than running one. To create a test, you need to have a problematic HeaderDoc comment block and declaration. After you have these, perform the following steps:

1. Create a minimal test case header or script that contains only the declaration that causes the problem (and the class surrounding it, if applicable).
2. Rename all classes, functions, variables, and so on to have a name that is unique across the entire test set.

> **Important:** The HeaderDoc test suite does not reset the symbol name cache between tests so that certain tests can verify the correctness of the API reference (`apple_ref`) collision handling code. This also means that you must name all functions, variables, and so on in a way that is unique across all of the tests. If you do not, your tests will behave differently when run individually than when run as a group.
>
> If you include the name of your test (with underscores for spaces) as part of every symbol name, your names will almost never collide with those in any other test. Be sure to do this for *all* names, including constants within enumerations.

3. Fix the bugs in HeaderDoc so that the code is parsed correctly.
4. Run the existing tests to make sure you didn't break anything else.
5. Type `headerDoc2HTML.pl -T create` to create a new test.
6. Choose a unique name for the test. The name for the test must be unique after spaces and special characters are replaced by underscores. For a list of existing tests, look in the subdirectories within the `testsuite` directory.

   In general, the names of tests in languages other than C or C++ should begin with the name of the programming language.
7. Choose a programming language for the test. (This must match the declaration.)
8. If you are creating a C language test, choose whether you are creating a C preprocessor test or a parser test.

   Parser tests are intended to test both the code parser and comment handling code. Most of the time, you should be writing a parser test. Parser tests are limited to a single declaration per test, however, and thus are unsuitable for determining whether a C preprocessor directive correctly modifies a declaration.

   C preprocessor tests are intended to test whether a C preprocessor macro correctly alters a declaration. Unlike a parser test, the C preprocessor test does not test the comment handling code. It takes two blocks of code. The first block is a series of C preprocessor macros that are all parsed. The second is a declaration to operate on. This declaration is parsed and information about that declaration is stored in the test results.

From this point on, the steps differ based on the type of test you choose.

Creating a Parser Test

To create a parser test, after completing the steps in "Creating a Test" (page 83), do the following:

1. Paste the HeaderDoc comment block, then press Control-D on a new line to end the comment.
2. Paste the code, then press Control-D on a new line to end the declaration.

3. Type in an explanatory message to describe how the test case differs from other similar tests, then press Control-D on a new line to end the message.
4. If you are overwriting an existing test case, HeaderDoc asks you to confirm that you intended to do so. In general, say no unless you made a mistake previously and are overwriting a test you just created.
5. Wait for HeaderDoc to build the test data.
6. Submit the test case and patches via bugreport.apple.com.

Creating a C Preprocessor Test

C preprocessor macro tests apply a series of C preprocessor macros to a declaration (which can be anything from a C preprocessor macro to a class).

To create a C preprocessor test, complete the steps in "Creating a Test" (page 83), then do the following:

1. Paste the block of C preprocessor macros, then press Control-D on a new line to end the list of macros.
2. Paste the declaration that the C preprocessor macros should modify, then press Control-D on a new line to end the declaration.
3. Type an explanatory message to describe how the test case differs from other similar tests, then press Control-D on a new line to end the message.
4. If you are overwriting an existing test case, HeaderDoc asks you to confirm that you intended to do so. In general, say no unless you made a mistake previously and are overwriting a test you just created.
5. Wait for HeaderDoc to build the test data.
6. Submit the test case and patches via bugreport.apple.com.

HeaderDoc Release Notes

The HeaderDoc Tools Suite consists of a series of Perl scripts and several small C helper applications that allows conversion of documentation embedded in header files in many languages into HTML and other output formats.

HeaderDoc 8 is the latest incarnation of the HeaderDoc tool and encompasses a series of versions:

HeaderDoc 8.0

HeaderDoc 8 is nearly a rewrite of HeaderDoc from the ground up. It incorporates the functionality of previous versions but also provides a number of new features, such as declaration syntax coloring/highlighting and an easier-to-use comment syntax. These features are described in "Major Features" (page 88).

HeaderDoc 8 adds a number of additional languages with various levels of support. These are described in "Languages Supported" (page 87).

HeaderDoc 8 also adds a number of new (optional) tags for convenience. These are described in "New Tags" (page 89).

HeaderDoc 8.5

HeaderDoc 8.5 adds a C preprocessor for more advanced header parsing. This is described in "Using the C Preprocessor" (page 69).

HeaderDoc 8.6

HeaderDoc 8.6 is a bug fix update to HeaderDoc 8.5, with a few minor features added.

HeaderDoc 8.7

HeaderDoc 8.7 adds support for Doxygen tags (@ form only) and adds support for IDL files. In addition, it includes a test suite (source distribution only) and contains numerous bug fixes.

HeaderDoc 8.8

HeaderDoc 8.8 adds support for AppleScript and Python, along with partial support for Tcl and Ruby. See "Troubleshooting" (page 107) for more information about limitations in Tcl and Ruby support.

HeaderDoc 8.8 also enhances the `resolveLinks` tool to support importing external cross-reference files.

Finally, HeaderDoc 8.8 bundles the regression test suite as part of the installation.

Languages Supported

HeaderDoc 8 supports many more languages than HeaderDoc 7. This table shows the various languages and the level of support.

Table A-1 HeaderDoc 8 Language Support

Language	HeaderDoc 7 support	HeaderDoc 8 support
AppleScript	no	yes (8.8)
C headers	yes	yes
C++	yes	yes
Objective C	yes	yes
C source code	no	yes
IDL	no	yes (8.7)
K&R C sources	no	yes
Java	no	yes *
JavaScript	no	yes *
Pascal	no	yes
PHP	sort-of (hack)	yes
Perl	no	yes **
Python	no	yes (8.8)
Ruby	no	yes (8.8)
Shell Scripts	no	yes **
Mach IPC Interface Definitions	no	yes

Note:

* Java and JavaScript support only functions and classes.

** Some scripting languages support only functions and subroutines.

Major Features

HeaderDoc 8 has a number of new features.

- Function/data type groupings
- Declaration syntax coloring
- New tagless syntax

  ```
  /*! This is a comment about what comes next */
  ```

- Support for HeaderDoc tags embedded in declarations
- Support for `//!` markup style for embedded HeaderDoc declarations
- Automatic linking of data types in declarations
- Improved C++ support (namespace/template/access)
- The `gatherheaderdoc` tool is now template based
- PHP support (and a bunch of other languages) now included without patching
- Support for linking to other methods and data types within the same file
- Comment stripper
- Support for exceptions
- Now warns if tagged parameters don't match declaration
- Optional warning if parameters are not tagged
- Improved warnings for other invalid content
- Man page output path (via XML)
- DTD for output validation
- Translation of HTML to XHTML using `xmllint` when using XML output
- Nested class handling
- Customizable date format
- C pseudoclass support (`typedef struct`)
- Better nested class support

- C++ constructors/destructors now sorted first in the list of class methods.
- The @ignore tag—allows you to remove matching tokens from declarations
- "Unsorted" flag
- Summary function and method lists (a mini-TOC)
- Automated detection of numbered lists
- Automatic handling of availability macros
- Improved overall appearance
- Includes a regression test suite

New Tags

This section attempts to list all of the new tags added in HeaderDoc 8 (some of which were actually available, but undocumented, in HeaderDoc 7).

`@classdesign`
: Text block describing the overall design of a class

`@coclass`
: String describing a class that this class was designed to work with

`@dependency`
: String describing a class upon which this class depends heavily

`@exception`
: String describing an exception thrown by a function/method/class

`@functiongroup`
: Tag for grouping functions and methods; this takes priority over the `@group` tag with respect to functions and methods.

`@group`
: Tag for grouping data, functions, and so on, thus changing the order in which they appear in the table of contents.

  (Note: the `@functiongroup` tag takes priority over the `@group` tag for functions.)

`@helper`
: String telling what helper classes this class uses

`@helps`
: For helper classes, string telling what sort of classes this class was designed to help

`@instancesize`
: Text block containing the size of an instance of this class

`@methodgroup`
: See `@functiongroup`.

`@ownership`
: String describing what class instantiates the current class (for example, I/O Kit nubs)

`@performance`
: Text block to describe performance characteristics of a class (for example, "This class is not appropriate for use in high-performance environments")

`@security`
: Text block to describe security considerations when using this class

`@superclass`
: Adds superclass info to a C pseudoclass; also can be used to cause members of the superclass to be merged into the subclass

`@throws`
: See `@exception`.

Additional Notes

This section lists known issues in HeaderDoc 8. We hope to improve in these areas in future versions. If you find issues not listed here, please file bugs.

- HeaderDoc 8 is somewhat slower than previous versions. This is because the entire parser has been rewritten from the ground up and now does a token-based parse of the input file.

  While this approach should significantly improve the correctness of output (colorizer bugs notwithstanding), it is doing a lot more work than before, and thus takes longer.

- The default color scheme generated by HeaderDoc matches Xcode coloring. There are a number of files supplied as alternative color schemes, ranging from pleasant to utterly hideous and blinking (used mainly for testing). Swap out your `headerDoc2HTML.config` file as desired.

- Although a minimal `gatherheaderdoc` template is built into the tool itself, the default template used by `gatherheaderdoc` is an actual file that comes preinstalled as `Xcode.app/Contents/Developer/usr/share/headerdoc/conf/com.apple.headerdoc.exampletocteplate.html` inside the Xcode app bundle (or `/usr/share/headerdoc/conf/com.apple.headerdoc.exampletocteplate.html` if you built

HeaderDoc yourself). The format for this template is described in "Advanced HeaderDoc Configuration and Features" (page 62). Also see "Example gatherheaderdoc Template" (page 66) for an example of the template format.

Late-Breaking Bugs

This section describes late-breaking bugs in HeaderDoc 8.9.

There are no known errata yet for HeaderDoc 8.9.

To keep up to date with the latest errata and bug fixes, join the headerdoc-dev mailing list.

Symbol Markers for HTML-Based Documentation

As HeaderDoc generates documentation for a set of header files, it injects named anchors (`<a name="marker"></a>`) into the HTML to mark the location of the documentation for each API symbol. This document describes the composition of these markers.

As you will see, each marker is self describing and can answer questions such as:

- What is the name of this symbol?
- What type of symbol is this (for example function, typedef, or method)?
- Which class does this method belong to?
- What is the language environment: C, C++, Java, Objective-C?

With this embedded information, the HTML documentation can be scanned to produce API lists for various purposes. For example, such a list could be used to verify that all declared API has corresponding documentation. Or, the documentation could be scanned to produce indexes of various sorts. The scanning script could as well create hyperlinks from the indexes to the source documentation. In short, these anchors retain at least some of the semantic information that is commonly lost when converting material to HTML format.

The Marker String

A **marker** string is defined as:

```
marker := prefix '/' lang-type '/' sym-type '/' sym-value
```

A marker is a string composed of two or more values separated by a forward slash (`/`). The forward-slash character is used because it is not a legal character in the symbol names for any of the languages currently under consideration.

The prefix defines this marker as conforming to our conventions and helps identify these markers to scanners. The language type defines the language of the symbol. The symbol type defines some semantic information about the symbol, such as whether it is a class name or function name. The symbol value is a string representing the symbol.

Because the string must be encoded as part of a URL, it must obey a very strict set of rules. Specifically, any characters other than letters and numbers must be encoded as a URL entity. For example, the operator + in C++ would be encoded as `%2b`.

By default, the prefix is `//apple_ref`. However, the prefix string can be changed using HeaderDoc's configuration file.

The currently-defined language types are described in Table B-1 (page 93).

Table B-1 HeaderDoc API reference language types

`applescript`	AppleScript script
`c`	C header or source code
`cpp`	C++ header or source code
`doc`	Special namespace for documentation purposes. (Content should be considered unstructured except for the special forms noted in "Special API Reference Types in the doc Hierarchy" (page 97).)
`idl`	Interface Description Language file. **Note:** This value is the default value if no value for `IDLLanguage` is set in the configuration file. See "Basic HeaderDoc Configuration" (page 54) for more information.
`java`	Java header
`js`	JavaScript script **Note:** Some historical implementations used the string `javascript`.
`mig`	Mach Interface Generator interface description
`occ`	Objective-C header or source code
`pascal`	Pascal source code
`perl`	perl script
`php`	PHP script
`python`	Python script
`ruby`	Ruby script
`shell`	Bourne, Korn, Bourne Again, or C shell script

`tcl`	TCL script

The language type defines the language binding of the symbol. Some logical symbols may be available in more than one language. The `c` language defines symbols which can be called from the C family of languages (C, Objective-C, and C++).

Symbol Types for All Languages

The symbol types common to all languages are described in Table B-2 (page 94).

Table B-2 Symbol types for all languages

tag	struct, union, or enum tag
`econst`	an enumerated constant—that is, a symbol defined inside an enum
`tdef`	typedef name (or Pascal `type`)
`macro`	macro name (without '()')
`data`	global, instance, or file-static data
`func`	function name (without '()')

Symbol Types for Languages With Classes

`cat`
: Category name (Objective-C only).

`cl`
: Class name.

Note: In Perl, this is used for the names of packages, and thus the names may contain a double colon between parts of package names. For example:

```
//apple_ref/perl/cl/HeaderDoc::APIOwner
```

`clconst`
: Constant values defined inside a class. For example:

```
//apple_ref/java/clconst/ClassName/kConstantName
```

`clm`

Class (or static [in java or c++]) method.

Note: The formats for method names are described in "Objective-C (occ) Method Name Format" (page 96) and "C++/Java (cpp/java) Method Name Format" (page 97).

`data`

Instance data. For example:

```
//apple_ref/cpp/data/MyClass/MyVariable
```

`intf`

Interface or protocol name.

`intfcm`

Class method defined in a protocol

Note: The formats for method names are described in "Objective-C (occ) Method Name Format" (page 96) and "C++/Java (cpp/java) Method Name Format" (page 97).

`intfm`

Method defined in an interface (or protocol).

Note: The formats for method names are described in "Objective-C (occ) Method Name Format" (page 96) and "C++/Java (cpp/java) Method Name Format" (page 97).

`intfp`

Property defined in an interface (or protocol)

```
//apple_ref/occ/intfp/ClassName/PropertyName
```

`instm`

Instance method.

Note: The formats for method names are described in "Objective-C (occ) Method Name Format" (page 96) and "C++/Java (cpp/java) Method Name Format" (page 97).

`instp`
: Instance property. For example:

```
//apple_ref/occ/instp/ClassName/PropertyName
```

C++ (cpp) Symbol Types

`tmplt`
: C++ class template.

`ftmplt`
: C++ function template.

Note: The format for this type is described in "C++/Java (cpp/java) Method Name Format" (page 97).

`func`
: C++ scoped function (in other words, not extern 'C'); includes return type and signature as described in "C++/Java (cpp/java) Method Name Format" (page 97), but with no class name. For example:

```
//apple_ref/cpp/func/funcName/returnType/(argType,argType,argType)
```

Objective-C (occ) Method Name Format

The format for method names for Objective-C is:

```
class_name '/' method_name
e.g.: //apple_ref/occ/instm/NSString/stringWithCString:
```

For methods in Objective-C categories, the category name is *not* included in the method name marker. The class named used is the class the category is defined on. For example, for the `windowDidMove:` delegate method on `NSWindow`, the marker would be:

```
e.g.: //apple_ref/occ/intfm/NSObject/windowDidMove:
```

Objective-C Property Format

The format for an Objective-C protocol is:

```
class_name '/' protocol_name
e.g. //apple_ref/occ/instp/MyClass/MyProp
```

C++/Java (cpp/java) Method Name Format

The format for method names for Java and C++ is:

```
    class_name '/' method_name '/' return_type '/' '(' signature ')'
e.g.: //apple_ref/java/instm/NSString/stringWithCString/NSString/(char*)
```

For Java and C++, signatures are part of the method name; signatures are enclosed in parentheses. The algorithm for encoding a signature is:

1. Remove the parameter name; for example, `change (Foo *bar, int i)` to `(Foo *, int )`.
2. Remove spaces; for example, change `(Foo *, int )` to `(Foo*,int)`.

Interface Builder Bindings Format

The format for Interface Builder bindings is:

```
'binding' '/' class_name '/' binding_name
e.g. //apple_ref/occ/binding/myclass/mybinding
```

Special API Reference Types in the doc Hierarchy

In general, the `doc` hierarchy should be considered to be an opaque blob of content. You should not count on the structure of a `doc` API reference. However, there are a few special subtypes within the `doc` space that are significant and should be used only for the stated purpose.

- `uid`—A unique identifier for a document. You may use values generated by `uuidgen` here. All other values are reserved for use by Apple.
- `title:...`—A HeaderDoc-specific hierarchy for special identifiers generated from the name portion of a HeaderDoc comment. These are generated when:
    - A name is specified in the HeaderDoc comment that does not match any parsed name.

- A declaration is parsed that has no name (such as an anonymous enumeration) and the specified name contains spaces or other illegal characters.

The complete name for this reference part depends on the name of the original data type. For example, a typedef would be:

```
//apple_ref/doc/title:tdef/Whatever
```

Most of the time, if you see these in HeaderDoc output, it means that the name specified in a HeaderDoc comment is wrong.

- `enumconstant`, `functionparam`, `methodparam`, `defineparam`, `structfield`, `typedeffield`—Special reference types for fields within structures, parameters within functions, and so on. Appears at the relevant point in the documentation.

  These are rarely useful, but can be used in cases where, for example, a function has numerous parameters to link to a specific parameter in the list.

  The `enumconstant` field should only appear if a normal API reference marker (`econst`) does not, which means you are unlikely to actually see this marker type in practice.

- `anysymbol`—Valid in link requests *only*. A link request in this namespace causes the link resolver to look up the symbol by name instead of by API reference. For example, the link request:

  ```
  //apple_ref/doc/anysymbol/MyProject
  ```

  would match any of the following API references:

  ```
  //apple_ref/c/func/MyProject
  //apple_ref/cpp/instm/MyClass/MyProject/bool/(char*,int)
  //apple_ref/perl/data/MyProject
  //apple_ref/java/cl/MyProject
  ```

  And so on. If more than one of these symbols exists, it matches the nearest symbol in the hierarchy (as determined by the number of leading absolute path parts).

Using API References in the @link Tag

When an API reference marker appears in a comment, it looks exactly like a normal API reference marker, with one exception: at any point where a slash appears, it is legal to precede that slash with a backslash. The reason for this can be demonstrated by the following symbol marker:

```
    //apple_ref/cpp/instm/MyClass/MyMethod/void*/(char*,int)
```

Notice that `*/` appears in the symbol marker, which would ordinarily end a comment in many programming languages. To fix this, you would tweak the symbol to look like this:

```
/* ...
    @link //apple_ref/cpp/instm/MyClass/MyMethod/void*\/(char*,int) ... @/link
 ...
 */
```

This prevents the compiler from choking on the API reference marker. HeaderDoc transparently removes the backslash when processing the marker.

Using resolveLinks to Resolve Cross References

HeaderDoc includes a tool called `resolveLinks` (in `/usr/bin` or `Xcode.app/Contents/Developer/usr/bin` beginning in 8.8, in `/System/Library/Perl/Extras/`*`PERL_VERSION`*`/HeaderDoc/bin` in previous versions) that is used for resolving cross-references for you Wherever a cross-reference appears, a link is generated if the destination exists.

The `resolveLinks` tool processes an entire tree of content in two passes. In the first pass, it locates destination anchors. These destination anchors look like this:

```
<a name="//apple_ref/..."></a>
```

Each of these `name` values is an identifier for an API symbol. The format for these identifiers is specified in "The Marker String" (page 92).

In the second pass, `resolveLinks` searches for cross-references to these destinations. These cross-references can occur in one of two forms, depending on whether a destination is known to exist or not.

```
<a logicalPath="//apple_ref/..." href="path">foo</a>
<!-- a logicalPath="//apple_ref/..." -->
```

Each of these `logicalPath` values is then paired (if possible) with `name` values obtained during the first pass. If a destination exists for a cross-reference, `resolveLinks` inserts the relative path of the destination anchor in the cross-reference request's `href` attribute. The result is that the cross-reference anchor is now a valid link to the requested destination anchor.

If the link exists and the cross-reference request is in the form of a comment, the `resolveLinks` tool changes the cross-reference request from a comment into an anchor (link) tag. Similarly, if the destination does not exist, it changes the cross-reference from an anchor tag to a comment tag. The result is that there should never be any broken links.

For the most part, this process is transparent to you as a user. There are two exceptions, however: cross-references between document sets and cross-references using multiple API reference prefixes (such as `apple_ref`).

Resolving Conflicting API References

In general, API references should not conflict. However, if two symbols with identical names and types occur in different namespaces, it is possible to have a conflict when you link together documentation that contains both namespaces.

When this occurs, HeaderDoc makes a best effort attempt at choosing the right match. For each potential link destination, HeaderDoc examines the path of the file containing that anchor and counts the number of leading path parts that match between that path and the path of the file that contains the link request. Then, HeaderDoc chooses the destination with the most matching path parts. (In the event of a tie, HeaderDoc typically chooses the first destination parsed, but you should not count on this ordering.)

Using Multiple API Reference Prefixes

If you use multiple API reference prefixes in a single tree of output content and want to link it together using `resolveLinks`, you must tell `resolveLinks` to look for all of the prefixes you care about. There are two ways to do this:

- Run `resolveLinks` manually, specifying the `-r` flag for each prefix. For example:

  ```
  resolveLinks -r david_ref -r joe_ref /path/to/dir
  ```

- Specify a list of valid prefixes in your `headerDoc2HTML.config` file using the `externalAPIUIDPrefixes` option.

  Note: This configuration file is read by `gatherHeaderDoc`, not by `resolveLinks`. Thus, this configuration file setting affects the behavior of `resolveLinks` *only* when `resolveLinks` is run by `gatherHeaderDoc`, not when you run `resolveLinks` manually.

Using External Cross-Reference Files

Whenever `resolveLinks` processes a tree, it generates a cross-reference file for that content. By default, it saves this file as `/tmp/xref_out`, but you can change this with the `-x` flag for later use.

If you want to process a tree in read-only mode (without writing back changes to the tree itself), you can specify the `-n` (no write) flag. In this mode, it will generate a cross-reference output file, but will not modify the HTML input files.

Beginning in HeaderDoc 8.8, `resolveLinks` supports additional flags to take advantage of these cross-reference files. Typically, you would use some combination of the `-s`, `-S`, `-b`, and `-i` flags.

These three flags are interrelated in subtle ways. The purpose of the complexity is so that you can construct links between two folders in such a way that the links will be valid after the folders are put into their final location. To that end, the flags provide prefix stripping and prepending.

> **Note:** The `resolveLinks` tool does not actually install any directories. The paths you pass in are merely to tell `resolveLinks` where you intend to install the folders at a later time so that it can construct relative paths correctly.

It is easiest to explain these flags by providing an example of a common use case. You have two directories, A and B.

Current location	Final location
`/Users/myusername/A`	`/Library/WebServer/Documents/Tools/A`
`/Users/myusername/B`	`/Library/WebServer/Documents/Utilities/B`

To create these links, you would first generate cross-reference files for each folder like this:

```
resolveLinks -n -x /tmp/A.xrefs -b "$PWD/" "A"
resolveLinks -n -x /tmp/B.xrefs -b "$PWD/" "B"
```

The paths in the resulting cross-reference file are in the form `A/...` or `B/...`.

Important: The trailing slash on the `-b` flag is significant. For scripting compatibility with other tools such as `cp`, the `resolveLinks` tool treats a trailing slash as an indication that the contents of the directory `$PWD` (the current working directory) will be installed in that location (in this case, the A folder).

If you omit the trailing slash, it assumes you are going to be installing the entire `$PWD` directory there, so in this example, the paths in the cross-reference file would be in the form `myusername/A/...` and `myusername/B/...`.

In a similar fashion, if you wanted to install the contents of the `"A"` folder and not the folder itself, you could pass `"$PWD/A/"` as the `-b` value.

Next, you must actually resolve the links. This is where the other flags come into play.

```
resolveLinks -b "$PWD/" -s /tmp/B.xrefs -S "/Library/WebServer/Documents/Utilities/"
 -i "/Library/WebServer/Documents/Tools/" "A"

resolveLinks -b "$PWD/" -s /tmp/A.xrefs -S "/Library/WebServer/Documents/Tools/"
-i "/Library/WebServer/Documents/Utilities/" "B"
```

You can pass in multiple pairs of `-s` and `-S` flags (up to a maximum of 1024) for additional flexibility. For each seed file, you must first use the `-s` flag to specify the location of the seed file itself, then use the `-S` flag to specify the location where the content described by that seed file will eventually be installed.

Important: The order of these flags is significant. The `-S` path modifies *only* the preceding `-s`. Thus, it *must* follow the `-s` seed file that it modifies without any intervening `-s` flags.

Alternatively, if you pass a `-S` flag *before* the first *-s* flag, `resolveLinks` treats it as the default base location for any subsequent `-s` flags (except those that are followed explicitly by a `-S` flag of their own). Be sure to provide comments in your scripts when doing this, however. If someone comes along later and adds a second pair of `-s` and `-S` with the order similarly reversed, the results will be confusing, to say the least.

In addition to seed file paths, you should also use the `-i` flag to tell `resolveLinks` where the folder you are processing will eventually be installed. Note that as before, the `-b` flag determines what portion to strip from each path in the folder you are processing, and that the trailing slash in the `-b` flag is significant here as well.

In effect, you can think of the flags like this:

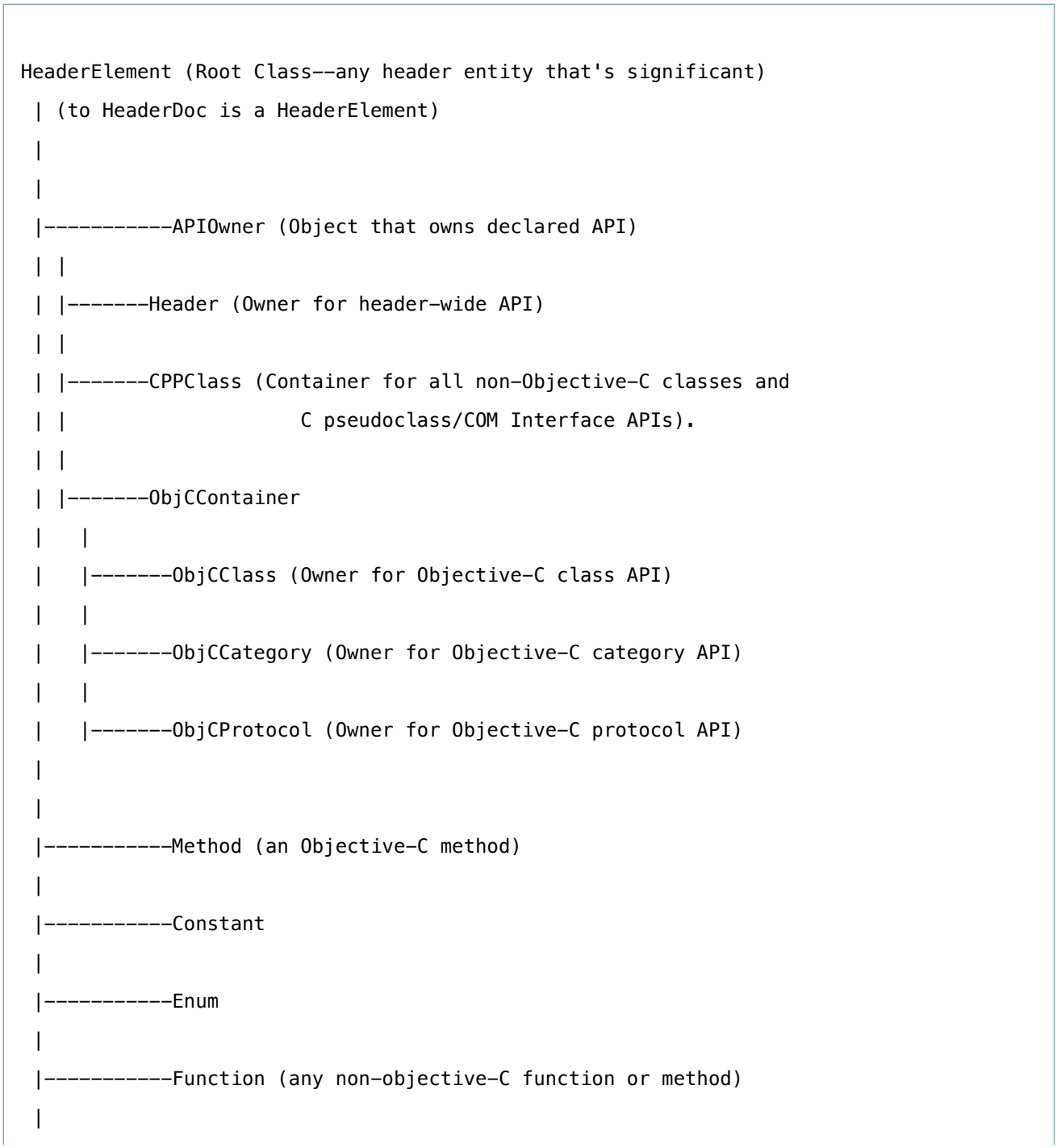
- `-b`—Strips off leading path parts from the folder you are *currently* processing. The last path part is stripped *only* if followed by a trailing slash.
- `-i`—Adds leading path parts to the folder you are *currently* processing (representing the proposed final install location).

- `-S`—Adds leading path parts to folders processed *previously* and imported from a seed file (representing their proposed final install locations).

Finally, if desired, you can pass the `-a` flag to tell `resolveLinks` to use absolute paths instead of relative paths when linking to the content described by a particular seed file or by all seed files. Like the `-S` flag, if passed before the first `-s` flag, the `-a` flag modifies the linking behavior globally. Otherwise, it modifies only the linking behavior for the preceding `-s` flag.

For more information, see the manual page for `resolveLinks`.

HeaderDoc Class Hierarchy



```
 |-----------MinorAPIElement (parameter, members of structs)
 |
 |-----------PDefine
 |
 |-----------Struct (for both structs and unions)
 | |
 | |-------Var (subclass of Struct so that it can contain fields)
 |
 |-----------Typedef


DocReference (Another root class. Used by gatherHeaderDoc to store
              information about documentation framesets within an
              input folder. The script uses this information to
              construct a top-level table of contents with links
              to each frameset.)

ParseTree (Token tree instantiated from BlockParse.pm.)

ParserState (Parser state instance instantiated from BlockParse.pm and stored
             in certain tokens within a ParseTree instance.)

IncludeHash (a simple data structure for storing information about a
             #include directive.)
```

In addition to the classes shown above, the `headerdoc2html/headerDoc2HTML.pl` script also uses the non–object-oriented modules `Utilities.pm`, `ClassArray.pm`, and `BlockParse.pm`. Most class instances are instantiated from `headerdoc2html/headerDoc2HTML.pl` based on the results of a call to `blockParse`.

The `ParseTree` class is instantiated in the block parser itself. It contains a token tree and a set of operations on that tree (print the tree, return a text or html representation of the tree, walk the parse tree for parameters, walk the parse tree for embedded HeaderDoc markup, and so on).

The `ParserState` class is also instantiated in the block parser. It contains only three methods (`new`, `_initialize`, and `print`), and is primarily just a giant hash with some pre-defined values.

The `IncludeHash` class is essentially just a simple data structure to handle basic information about #include directives. It has two methods (`new` and `_initialize`).

The `gatherHeaderDoc` tool uses an external program, `resolveLinks`, to convert special "link request" comments into links to other files in the directory being processed. This tool (written in C) resides in the `bin` directory within the HeaderDoc modules directory.

HeaderDoc uses `xmllint` (from libxml) to convert HTML into XHTML when generating XML output. HeaderDoc also uses `hdxml2manxml` and `xml2man` from the MPGL suite to generate man pages.

Troubleshooting

This chapter explains how to troubleshoot HeaderDoc issues, including explanations of error messages, in the form of a Q&A list.

This troubleshooting guide assumes that you are running the latest version of HeaderDoc (currently 8.8). If not, you should first upgrade to the latest version and see if your problem goes away.

You can get the latest version of HeaderDoc on the Apple Open Source website: http://www.opensource.apple.com/. Choose the most recent version of OS X or Xcode, then search for `headerdoc` on the page, and click the download link in the rightmost column. Except as described below, HeaderDoc should work correctly on earlier versions of OS X. If it does not, please file a bug.

Common Error Messages

Q: When running gatherheaderdoc, I get the error message "Using the -d flag requires the HTML::FormatText module. To install it, type: sudo cpan HTML::FormatText".

A: Doc set generation requires two Perl modules that are not present in OS X prior to 10.7. To install those modules, log in as an admin user, then type:

```
sudo cpan HTML::FormatText
sudo cpan HTML::TreeBuilder
```

Enter your admin password when prompted. For more information, see the `cpan` man page and http://www.cpan.org/.

Q: When running gatherheaderdoc, I get an error from something called resolveLinks that says "I/O error: encoder error". What's going on?

A: You have a header file that was not written in UTF-8. Change the encoding for that file by adding an `@encoding` or `@charset` entry within the `@header` tag.

Beginning in HeaderDoc 8.8, HeaderDoc attempts to detect this automatically. If you see this error in HeaderDoc 8.8 or later, please file a bug and enclose a copy of the header.

Q: HeaderDoc keeps warning me that my LibXML2 version is too old. How do I fix this problem?

A: Obtain a more recent version of LibXML2 from http://www.xmlsoft.org.

Q: I'm trying to do an @link to a method, but HeaderDoc insists that myMethodname%58 could not be found.

A: Beginning in HeaderDoc 8.5, you should use colons in the names of methods in @link tags, rather than replacing them with %58.

Q: HeaderDoc is choking on classes with multiple inheritance.

A: Update to HeaderDoc 8.5.

Q: Why isn't HeaderDoc doing C preprocessing? I thought you said this version did.

A: It does, but you have to specify an additional flag, `-p`, to invoke this behavior.

Q: The C preprocess keeps including the wrong files.

A: HeaderDoc has no way of knowing the final installed location of header files. To work correctly, it depends on all header files having a unique name. Rename your header files so that no two files have exactly the same name.

Q: I keep getting the error "Name being changed (oldname -> newname)."

A: This is usually caused by one of the following:

1.Multiple @discussion blocks. Remove one of them.

2.An extra preprocessor macro token after the close parenthesis in a function declaration. HeaderDoc thinks you are writing a K&R C declaration. Either use @ignore to ignore the token or explicitly mark up the preprocessor macro and enable C preprocessing.

Q: HeaderDoc says "Can't open <filename> for availability macros."

A: Your installation is likely missing the `Availability.list` file. In Xcode 4.3 and later, it lives in `/usr/share/headerdoc/`. In OS X v10.6 and v10.7 with Xcode 4.2.1 and earlier, it lived in `/System/Library/Perl/Extras/version/HeaderDoc`. In OS X v10.5 and earlier, it lived in `/System/Library/Perl/version/HeaderDoc`.

Q: I'm getting the error "Conflicting declarations for function/method ($name1) outside a class. This is probably not what you want."

A: As it says, you have two functions that are not class members, but have the same name (or you forgot to put HeaderDoc markup on the enclosing class). This is legal in C++ but is discouraged because the apple_ref syntax does not provide a uniqueness guarantee in these instances. HeaderDoc tries to fudge this by appending a signature when it sees this situation, but as a general rule, you should not rely on this behavior if you care about apple_ref markup.

Q: HeaderDoc is spewing warnings about "Parsed parameter <blah> not found in declaration of function/method/typedef <blah>."

A: Chances are, you made a typographical error when adding `@param` or `@field` markers in the HeaderDoc comment. Check your spelling carefully and remember that capitalization matters.

Q: HeaderDoc keeps saying "Tagged parameter <blah> not found in declaration of function/method/typedef <blah>."

A: You turned on the strict parameter/field checking with the `-t` flag. Turn it off if you don't want those warnings.

Q: HeaderDoc says "Braces/class braces/parentheses/square braces do not match. We may have a problem."

A: This usually means exactly what it says. If you are depending on a C preprocessor macro to make braces match, you should try to avoid doing so. If you cannot avoid this, make sure you enable C preprocessing and add HeaderDoc markup to the macro definition.

Q: HeaderDoc says "End of parse tree reached while searching for matching definition".

A: This is generally caused by either placing a HeaderDoc comment immediately prior to a close curly brace or by placing the wrong HeaderDoc type tag in the comment (such as preceding a `typedef` with an `@function` comment).

Q: HeaderDoc says "No matching declaration found. Last name was <blah>."

A: This is generally caused by either placing a HeaderDoc comment immediately prior to a close curly brace or by placing the wrong HeaderDoc type tag in the comment (such as preceding a `typedef` with an `@function` comment).

Q: I'm getting the error "Unable to process #define macro "<name>."

A: Please file a bug.

Q: HeaderDoc says "WARNING: multiple matches found for symbol "<blah>." Only the first matching symbol will be linked."

A: You have multiple symbols with the same name (possibly in different files, or possibly different types—for example a function and a `#define`). HeaderDoc has no way to know which of those two or more "myname" symbols you're talking about when you say `@link myname`. To fix this problem, look in the HeaderDoc-generated HTML for the desired destination. Find the name anchor that looks like `<a name="//apple_ref/...">` and instead of just giving the name, give the entire contents of that anchor.

Q: HeaderDoc says 'WARNING: no symbol matching "<blah>" found. If this symbol is not in this file or class, you need to specify it with an api ref tag (e.g. apple_ref).'

A: You may not be processing all of the needed files at once, or HeaderDoc may be feeling cranky. In any case, to fix this problem, look in the HeaderDoc-generated HTML for the desired destination. Find the name anchor that looks like `<a name="//apple_ref/...">` and instead of just giving the name, give the entire contents of that anchor.

Q: HeaderDoc issues the warning "WARNING: resolveLinks not installed. Please check your installation."

A: Be sure you are installing correctly. First, type "`make`", then "`make realinstall`".

Q: HeaderDoc says "WARNING: Unexpected headerdoc markup found in <blah> declaration."

A: Chances are, you followed one HeaderDoc comment with another HeaderDoc comment without anything in-between.

Q: HeaderDoc warns "Unterminated @link tag (starting field was: @link...)."

A: If you are using JavaDoc-style @link tagging (`{@link symbol Link Text}`), don't forget the close curly brace. If you are doing HeaderDoc-style @link tagging (`@link symbol Link Text @/link`), don't forget the `@/link`.

Q: HeaderDoc said "Parser bug: empty outer type."

A: This is probably a bug unless you're doing something really weird with preprocessor directives that violate the normal C syntax rules (in which case you should either @ignore the extraneous tokens or enable C preprocessing). In general, though, you should probably file a bug.

Q: HeaderDoc keeps saying "Objective-C method found outside a class or interface (or in a class or interface that lacks HeaderDoc markup)."

A: Make sure you properly tagged the enclosing class or interface declaration.

Q: HeaderDoc keeps saying "Unable to find parse tree. Please file a bug."

A: This should not happen; please file a bug.

Q: HeaderDoc keeps saying "Couldn't find parser state. Using slow method."

A: If you have a class that starts with a preprocessor token (such as `DeclareStructors(MyClass)` or similar), this will break things badly. There are two solutions. The easiest solution is to add `@ignorefuncmacro DeclareStructors` (or whatever the macro name happens to be) in your `@header` declaration.

An alternative fix is to make sure that you are processing the header file that contains the macro at the same time as you process the class. Enable C preprocessing with the `–p` flag. Finally, add a HeaderDoc comment before the macro definition.

If this problem is not caused by use of a macro, please file a bug. This fallback case should not affect output, however.

Q: HeaderDoc keeps saying 'Could not determine include file name for "#include FW(Carbon,CarbonEvents.h)"' or similar.

A: Ideally, you should use a standard include file syntax. If that is not possible, you should enable the C preprocessor with the `-p` flag, include the file containing the FW macro on the command line, and add HeaderDoc markup to that macro.

Q: HeaderDoc says "Unknown regexp delimiter '...'. Please file a bug.

A: This should not happen; please file a bug.

Q: HeaderDoc keeps saying "Unknown keyword <blah> in block-parsed declaration".

A: Make sure that the header compiles correctly with `gcc`. If it does, please file a bug.

Other error messages generally fall into one of two categories: self-explanatory errors (such as "Unknown tag @whatever in function comment") or utterly unintelligible (such as "Parser bug: empty outer type"). In the case of the former, please fix the appropriate declaration. In the case of the latter, pleas file a bug. Which brings us to the last question....

Q: How do I file a bug?

A: Before filing a bug, you should subscribe to the HeaderDoc-dev mailing list on lists.apple.com. Ask if anyone else has seen the problem. If not, you should file a bug. To subscribe, visit http://lists.apple.com/mailman/listinfo/headerdoc-dev.

If you are an ADC member with access to bugreport.apple.com, please file a bug through that mechanism. The correct component is "HeaderDoc", with version "Darwin".

Unexpected Behavior

Q: Some of my Ruby, JavaScript, or Tcl scripts are missing content.

A: HeaderDoc 8.8 did not parse regular expressions in any language other than Perl.

In most languages, this makes no difference because regular expressions are stored in strings and obey normal string parsing rules. In Python, this makes no difference because its parser is based solely on indentation depth. This leaves three languages in which regular expressions can cause problems: Ruby, JavaScript, and Tcl.

Although most regular expressions do not cause problems, expressions that match literal braces, quotation marks, parentheses, and other symbols may confuse HeaderDoc because it is unaware that they are appearing in the context of a regular expression.

The recommended solution is to upgrade to HeaderDoc 8.9, which should fix the problem. If upgrading is not possible, you can work around this issue by adding a throwaway regular expression in a previous or subsequent line that contains the necessary matching open or close brace, parenthesis, or quotation mark to work around the problem. (This extra regular expression must *not* be in a comment.)

Q: I'm seeing multiple copies of my functions/typedefs/defines/*. Why?

A: You probably specified a name in the `@function` tag (or `@typedef` or...) that was different from the actual name. Delete the incorrect name or pass the -N flag to HeaderDoc, which tells it to globally ignore any names specified in the HeaderDoc markup (HeaderDoc 8.8 and later).

Q: I'm still seeing multiple copies of a typedef, but with different names.

A: HeaderDoc, by default, also generates an entry for "tag names" and for every type name. You can remove the tag names by specifying the -O (outer names only) flag. In the following example, the tag name is `mystruct`, and the type name (a.k.a. the "outer name") is `mystruct_t`:

```
typedef struct mystruct {int a;} mystruct_t;
```

Q: Why does my function/method/type/variable/class/* have a name that appears to include an entire paragraph of the discussion?

A: One of two things is wrong. Either you included multiple words after the `@function/@typedef/@whatever` and also included an `@discussion` tag or you began a multi-line declaration at the end of the `@function/@typedef/@whatever` line. Don't do this. See "Multiword Names" (page 22) for more information.

Q: A bunch of my `functions/typedefs/*` are being linked together with "See Also" attributes. What's up?

A: You probably marked a `typedef` with an `@function` comment (or some other incorrect pairing). HeaderDoc will assume that you knew what you were doing and will keep looking through the code until it finds whatever was requested (a function in this case). Everything in-between will get linked together. The purpose for this is primarily to allow you to mark `@typedef` for a `struct` followed by a `typedef`, but it is useful in other situations as well. Fix the incorrectly matched comment, and the problem should go away.

Q: Why am I not getting links when I add `@link` tags?

A: There are several possible reasons:

1.If you used `apple_ref` markup, you may have made a typo.

2.If you used a symbol name that did not exist, you would get a warning when running `headerdoc2html` and no link will be generated.

3.The `@link` tag only inserts a link request into the HTML. To turn this into an actual link, you must run `gatherheaderdoc` (which in turn runs `resolveLinks` to create the links). Until you run gatherHeaderDoc, all you will see in the HTML are a bunch of specially-formatted comments.

Q: Every time I run `gatherheaderdoc`, all of the spaces in my declarations go away. What's going on?

A: This is a bug in `libxml2` that is fixed in more recent versions. Please visit http://www.xmlsoft.org to obtain a more recent version (or upgrade to OS X v10.4 or later).

Other Issues

Q: I'm confused. Where can I get help?

A: The best place to get help is the HeaderDoc-dev mailing list. To subscribe, go to http://lists.apple.com/mailman/listinfo/headerdoc-dev.

Document Revision History

This table describes the changes to *HeaderDoc User Guide* .

Date	Notes
2012-02-16	Updated for Xcode 4.3 and improved the tags sections.
2011-06-06	Added explanation of new features.
2010-08-10	Updated errata.
2009-11-17	Added additional 8.7 (Snow Leopard) flags.
2009-06-23	Fixed minor typos.
2009-05-08	Updated to cover new flags and tags in HeaderDoc 8.7 (OS X v10.6). Corrected apple_ref types.
2008-04-08	Updated for OS X v10.5.
2006-11-07	Significantly restructured the tagging chapter.
2006-10-03	Made minor corrections.
2005-04-29	Changed title from "HeaderDoc Unfettered."
2004-11-02	Added troubleshooting chapter and information about HeaderDoc 8.5. Added revision history, title change.
2004-06-28	Updated for HeaderDoc 8.
2004-04-01	Translated from original HTML version and updated for HeaderDoc 8 public beta

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010