HeaderDoc Unfettered

May 27, 2004

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

Simultaneously published in the United States and Canada.

Contents

Tables and Listings

Introduction to HeaderDoc: API Documentation From Header Files

This document describes how to use the HeaderDoc tool. It also explains how to insert HeaderDoc comments into your headers and other files.

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.

In addition to traditional HeaderDoc markup, HeaderDoc 8 supports JavaDoc markup. HeaderDoc 8 also supports a number of languages: Bourne shell (and Korn and Bourne Again), C Headers, C source code, C shell, C++ headers, Java, JavaScript, Mach MIG definitions, Objective C/C++ headers, Pascal, Perl, and PHP. Most of these languages (besides C/C++/ObjC/Pascal) support documenting only functions or subroutines.

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 "Using gatherHeaderDoc" (page 37).

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

Finally, 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`.

Organization of this Document

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

- "HeaderDoc Tags" (page 9) explains how to add HeaderDoc markup to header (and source code) files.

- "Using HeaderDoc" (page 33) explains the syntax for the HeaderDoc command-line tool itself.
- "Using gatherHeaderDoc" (page 37) explains how to use gatherHeaderDoc to produce landing pages and cross-linked trees of related documentation.
- "Using the MPGL Suite" (page 43) exmplains how to use the Manual Page Generation Language (MPGL) tool suite.
- "Configuring HeaderDoc" (page 51) explains the HeaderDoc configuration file.
- "Symbol Markers for HTML-Based Documentation" (page 55) describes the symbol markers used by HeaderDoc and various other utilities to provide linking functionality.
- "HeaderDoc Class Hierarchy" (page 59) describes the class hierarchy of the HeaderDoc tool itself.

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 below, the "Fields" column indicates the number of textual fields each type of tag takes.

Introduction to HeaderDoc Comments and Tags

HeaderDoc comments 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.

Historically, HeaderDoc tags also required the addition of a tag that announces the type of API being commented (`@function`, below). Beginning in HeaderDoc 8, this tag became optional.

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

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 "C Pseudoclass Tags" (page 23).

```
/*!
 @class ClassName
 This is a comment about ClassName.
*/
```

```
typedef struct foo {...};
```

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, `\@`.

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 comment 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_timeLatency before next read. */
```

You can also use leading asterisks on each line of a multiline comment:

```
 /*!
 * @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 carriage returns 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 a discussion tag, like this:

Listing 2-1 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 after the name.

Listing 2-2 Example of multiword names using multiple lines

```
/*!
 * @enum example enum
 * This is a test, this is only a test.
 *
 * Because the discussion contains multiple lines, 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.

Tags for Frameworks

Framework documentation should be inserted into a file ending in `.hdoc`. Running HeaderDoc on this file generates a documentation tree with special hidden markup that gatherHeaderDoc will insert into the appropriate place within your TOC template (or at the top of the built-in template).

Tag	Example	Identifies	Fields
`@framework`	`@framework Kernel Framework`	The name of the framework.	1
`@abstract`	`@abstract In-kernel device driver framework`	A short string that briefly describes a framework. This should not contain multiple lines (at least for the default template) for aesthetic reasons. Save the detailed descriptions for the `@discussion` tag.	1
`@discussion`	`@discussion The kernel framework contains functions useful to in-kernel device drivers.`	A detailed description of the framework. This may contain multiple paragraphs, and can contain HTML markup.	1

Tags for All Headers

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.

The value you give for the `@header` tag serves as the title for the HTML pages generated by `headerDoc2HTML`. The text you associate with the `@header` tag is used for the introductory page of the HTML website the script produces.

In general, however, you will not specify a filename in the `@header` tag, and will simply let HeaderDoc substitute the filename. 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 tags in the table below (with the exception of `@header`, which must be at the start of a comment block) can be used in any comment for any data type, function, header, or class.

Note: Some tags are marked as inline-compatible. This means that they can be used in the middle of a text container such as `@discussion` or `@abstract`, and that their contents will appear within the existing text flow.Non–inline-compatible tags terminate the existing text container and create a new one.

Tag	Example	Identifies	Fields
@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.	1
@availability	`@availability 10.3 and later`	A string that describes the availability of a function, class, and so on.	1
@copyright	`@copyright Apple`	Copyright info to be added to each page. This overrides the config file value and may not span multiple lines.	1
@deprecated	`@deprecated in version 10.4`	String telling when the function, data type, etc. was deprecated.	1
@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
@header	`@header Repast Manager`	The name under which the API is categorized. Leave the name blank to just use the header filename. The following subtags are available:	0/1

Tag	Example	Identifies	Fields
		`@CFBundleIdentifier`	
			STRING which kernel subcomponent or loadable extension contains this header
		`@compilerflag`	
			TERM/DEFINITION compiler flag that should be set.
		`@flag`	
			TERM/DEFINITION same as `@compilerflag`.
		`@ignore`	
			token to ignore.
		`@preprocinfo`	
			BLOCK: description of behavior when `#define DEBUG` is set, and so on.
		`@related`	
			TERM/DEFINITION indicates another header that is related to this one. You may use multiple `@related` tags.
		Explanation of types:	
		■ *STRING: single string, like @abstract*	
		■ *TERM/DEFINITION: <name> <description>, like @enum*	
		■ *BLOCK: block of text like @discussion*	

Tag	Example	Identifies	Fields
@link	@link apple_ref//c/func/function_name link text goes here @/link *or* @link function_name link text goes here @/link	Allows you to insert a link request for an API ref. If the link target is part of the same .h file, you can do this by using only the name of the function or data type. If it is in a separate file (or if there are multiple matches for a given name), you must explicitly specify which API ref to use. Because the headerDoc2HTML script does not know the actual target for these links, it inserts comments into the output. You must then run gatherHeaderDoc to actually turn those comments into working links. **This tag is an inline-compatible tag.**	1
@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.	1
@namespace	@namespace BSD Kernel	String describing the namespace in which the function, data type, etc. exists.	1
@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 that this tag does not automatically insert <pre> or <tt>. You may wrap it with whatever formatting you choose.* **This tag is an inline-compatible tag.**	block
@updated	@updated 2003-03-14	The date at which the header was last updated.	1
@version	@version 2.3.1	the version number to which this documentation applies.	1

Listing 2-3 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"
*/
```

Tags Common to All API Types

The `@abstract`, `@updated`, and `@discussion` tags can be used within any of the type-specific tags below. For example:

```
/*!
 @enum Beverage Categories
 @abstract Constants to group beverage types.
 @discussion These constants (such as kSoda, kBeer, etc.) can be used...
 @updated 2003-04-15
*/
```

They are not required within any HeaderDoc comment, but can improve the formatting of the HTML output, and can help automate the importation of comments into the Inside Mac documentation database.

Tags for All Languages

Availability Macro Tags

Tag	Identifies	Fields
`@availabilitymacro`	The name of the availability macro and a string describing it. If the macro name appears in the declaration of any later function, class, method, or data type, the string will be added to its documentation as an availability attribute. See "Automatic Tagging" (page 12) for more information.	2

Listing 2-4 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 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.

Constant Tags

Tag	Identifies	Fields
`@const` or `@constant`	Name of the constant.	1

Listing 2-5 Example of @const tag

```
/*!
@const kCFTypeArrayCallBacks
@discussion Predefined CFArrayCallBacks structure containing a set of
callbacks appropriate...
*/
const CFArrayCallBacks kCFTypeArrayCallBacks;
```

#define Tags

Tag	Identifies	Fields
`@defined`	Name of the macro.	1

Note: Function-like defines with curly braces may also use `@function`. This option is included for legacy compatibility and has no effect on the resulting output.

Listing 2-6 Example of @defined tag

```
/*!
@defined TRUE
@discussion Defines the boolean true value.
*/
#define TRUE 1
```

For more usage examples, see the `ExampleHeaders` folder that accompanies the HeaderDoc distribution.

Enum Tags

Tag	Identifies	Fields
`@enum`	The name of the enumeration. This is the enum's tag, if it has one. Otherwise, supply a name you want to have the constants grouped under in the documentation.	1
`@constant`	A constant within the enumeration.	2

Listing 2-7 Example of @enum tag

```
/*!
 @enum Beverage Categories
 @discussion 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.
*/
enum {
 kSoda = (1 << 6),
 kBeer = (1 << 7),
 kMilk = (1 << 8),
 kWater = (1 << 9)
}
```

Function Tags

Tag	Identifies	Fields
`@function`	The name of the function or macro.	1
`@param`	Each of the function's parameters.	2
`@result`	The return value of the function. Don't include if the return value is void or OSERR	1
`@throws`	Include one `@throws` tag for each exception thrown by this function (in languages that support exceptions).	1
`@templatefield`	Each of the function’s template fields (C++).	2

Listing 2-8 Example of @function tag

```
/*!
 @function ConstructBLT
 @discussion Creates a Sandwich structure from the supplied arguments.
 @param b Top ingredient, typically protein-rich.
 @param l Middle ingredient.
 @param t Bottom ingredient, controls tartness.
 @param mayo A flag controlling addition of condiment. Use YES for
condiment,
 HOLDTHE otherwise.
 @throws peanuts
 @templatefield K The type of BLT to be generated (I want a BLT float)
 @result A pointer to a Sandwich structure. Caller is responsible for
 disposing of this structure.
*/
Sandwich *ConstructBLT<K>(Ingredient b, Ingredient l, Ingredient t, Boolean
 mayo);
```

Function Group Tags

Tag	Identifies	Fields
`@functiongroup`	The name of the function group.	1

Listing 2-9 Example of @functiongroup tag

```
/*!
 @functiongroup Core Functions
*/
```

Function groups are not required, but they allow you to organize a large number of functions into near groupings. The `@functiongroup` tag remains in effect until the next `@functiongroup` tag.

If you need to put functions in different parts of the header into the same group, simply give them the same name (with the same capitalization, punctuation, spacing, etc.), and it will merge the two function groups into one.

Note that functions encountered before the first `@functiongroup` are considered part of the "empty" group. These functions will be listed before any grouped functions.

Struct and Union Tags

Tag	Identifies	Fields
`@struct / @union`	The name of the structure or union. (Also known as the struct or union's tag.)	1
`@field`	A field in the structure.	2

Listing 2-10 `Example of @struct` tag

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

Typedef Tags

Tag	Identifies	Fields
`@typedef`	The name of the defined type.	1
various	The tags that can appear after a "`@typedef`" tag depend on the definition of the new type. `@field` for typedef'd structs `@constant` for typedef'd enumerations `@param` for simple typedef'd function pointers `@callback`, `@param`, `@result` for typedef'd structs containing function pointers	

Listing 2-11 Typedef for a simple struct

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

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

Listing 2-12 Typedef for an enumeration

```
/*!
 @typedef TypedefdEnum
 @abstract Abstract for this API.
 @discussion Discussion that applies to the entire typedef'd enum.
 @constant kCFCompareLessThan Description of first constant.
 @constant kCFCompareEqualTo Description of second constant.
 @constant kCFCompareGreaterThan Description of third constant.
*/
typedef enum {
 kCFCompareLessThan = -1,
 kCFCompareEqualTo = 0,
 kCFCompareGreaterThan = 1
} TypedefdEnum;
```

Listing 2-13 Typedef for a simple function pointer

```
/*!
 @typedef simpleCallback
 @abstract Abstract for this API.
 @discussion Discussion that applies to the entire callback.
 @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.
*/
typedef long (*simpleCallback)(short inFirstParameter, unsigned long long
 *outSecondParameter);
```

Listing 2-14 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;
```

Variable tags

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

Tag	Identifies	Fields
`@var`	The name of the data member followed by the description.	2

Listing 2-15 Example of @var tag

```
/*! @var we_are_root TRUE if this device is the root power domain */

 boolwe_are_root;
```

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.

`Class` Tags

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

Listing 2-16 Example of @class tag

```
/*!
 @class IOFireWireDeviceInterface_t
*/
 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.

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.

Interface Tags

Tag	Identifies	Fields
`@interface`	The name of a COM interface.	1

Listing 2-17 Example of @interface tag

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

The `@class` tag causes the `typedef struct` that follows the headerdoc comment to be treated as a COM interface (which is essentially a C pseudoclass with a different name).

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.

Superclass Tags

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

Listing 2-18 Example of @superclass tag

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

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 that in order for this feature to work, both pseudoclasses must be processed at the same time.

Tags for C++ Headers

HeaderDoc processes a C++ header 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. This means you can use any of the tags defined for C headers within a C++ header. See "Tags for All Languages" (page 17).

For example, in a header that declares two classes, you may want to use the `@header` tag to provide a discussion explaining why these classes are grouped, and use the `@class` tags to provide discussions for each of the classes.

Once HeaderDoc encounters an `@class` tag (with accompanying class declaration) in a C++ header, it treats all comments and declarations that follow (until the end of the class declaration) as belonging to that class, rather than to the header as a whole. When HeaderDoc generates the HTML documentation for a C++ header, it creates one frameset for the header as a whole, and separate framesets for each class declared within the header.

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

Within a C++ class declaration, HeaderDoc allows some additional tags, as describe below.

Conventions

Tags, depending on type, can introduce either one or two fields of information:

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

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

Additional Tags for C++ Class Declarations

Within a C++ class declaration, HeaderDoc understands all the tags for C headers, along with some new ones which are listed in the following sections.

Class Tags

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

Following the `@class` 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 2-19 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 2-20 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...
 @throws foo_exception
 @throws bar_exception
 @updated 2003-03-15
*/
class IOCommandGate: public IOEventSource
{
...
}
```

Classes have many special tags associated with them for convenience. These include:

Tag	Disposition
`@classdesign`	BLOCK: description of any common design considerations that apply to this class, such as consistent ways of handling locking or threading
`@coclass`	TERM/DEFINITION: class with which this class was designed to work
`@dependency`	STRING: external resource that this class depends on (such as a class or file)
`@helper` or `@helperclass`	TERM/DEFINITION: helper classes used by this class
`@helps`	STRING: if this is a helper class, short description of classes that this class was designed to help
`@instancesize`	STRING: the size of each instance of the class
`@ownership`	BLOCK: description of ownership model to which this class conforms, e.g. MyClass objects are owned by the MyCreatorClass object that created them
`@performance`	BLOCK: description of this class's special performance characteristics, e.g. this class is optimized for the Velocity Engine, or this class is strongly discouraged in high-performance contexts

Tag	Disposition
`@security`	BLOCK: security considerations associated with the use of this class
`@superclass`	STRING: override superclass name---see note below.

Explanation of types:

- *STRING: single string, like @abstract*
- *TERM/DEFINITION: <name> <description>, like @enum*
- *BLOCK: block of text like @discussion*

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.

Function Tags

For member functions, use the `@function` tag (described in "Function Tags" (page 19)) or the `@method` tag (which behaves identically for C++ methods).

Template Tags

For C++ template classes, if you want to document the template type parameters, you should use the `@templatefield` tag. You should also be sure to define the class using `@template` instead of `@class`.

The `@templatefield` tag can also be used to document template parameters for C++ template functions.

Tag	Identifies	Fields
`@templatefield`	The name of the parameter followed by the description.	2

Listing 2-21 Example of @templatefield tag

```
/*! @templatemystackclass
 @templatefieldTthe data type stored in this stack */
```

```
template <T> class mystackclass
```

For more usage examples, see the `ExampleHeaders` folder that accompanies the HeaderDoc distribution.

Tags for Objective-C Headers

Tags for Objective-C Headers

Introduction

HeaderDoc processes a Objective-C header in much the same way that it does a C header. In fact, until HeaderDoc encounters a class declaration in an Objective-C header, the processing is identical. This means you can use any of the tags defined for C headers within an Objective-C header. See "Tags Common to All API Types" (page 17).

For example, in a header that declares two classes, you may want to use the `@header` tag to provide a discussion explaining why these classes are grouped, and use the `@class` tags to provide discussions for each of the classes. Within the 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.

Processing Objective-C Classes

Once HeaderDoc encounters an `@class` tag (with accompanying declaration) in an Objective-C header, it treats all comments and declarations that follow—until the end of the class declaration—as belonging to that class, rather than to the header as a whole. When HeaderDoc generates the HTML documentation for an Objective-C header, it creates one frameset for the header as a whole, and separate framesets for each class declared within the header.

Processing Objective-C Protocols

HeaderDoc processes Objective-C protocol declarations similarly to class declarations. The documentation for the protocol and the methods it declares are grouped in their own frameset, which is accessed from the documentation for the header that contains the protocol.

Processing 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 a Objective-C class, protocol, or category declaration, HeaderDoc allows the `@method` tag, as describe below.

The @class, @protocol, and @category Tags

In Objective-C, class and protocol declarations are quite similar, and consequently HeaderDoc's `@class` and `@protocol` tags are parallel in their usage.

Class and Protocol Tags

Tag	Identifies	Fields
`@class`	The name of the class.	1
`@category`	The full name of the category, as declared in the header. For example, "MyClass(MyCategory)". HeaderDoc uses the "MyClass" portion of the name to identify the associated class.	1
`@protocol`	The name of the protocol.	1

Following these tags, you typically provide introductory information about the purpose of the class, protocol, or category. 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.

Listing 2-22 Documentation tagged as abstract and discussion

```
/*!
 @class NSPrinter
 @abstract An NSPrinter object describes a printer's capabilities.
 @discussion An NSPrinter object describes a printer's capabilities, such
 as whether the printer can print in color and whether it provides a
particular font. An NSPrinter object represents...
*/
@interface NSPrinter: NSObject <NSCopying, NSCoding>
...
@end
```

Listing 2-23 Documentation included as single block of text

```
/*!
 @class NSPrinter
 An NSPrinter object describes a printer's capabilities, such as whether
 the printer can print in color and whether it provides a particular font.
 An NSPrinter object represents...
*/
@interface NSPrinter: NSObject <NSCopying, NSCoding$gt;
...
@end
```

The @method Tag

For methods declared in an Objective-C class, protocol, or category, use the `@method` tag.

Tag	Identifies	Fields
`@method`	The method name followed by the description.	2
`@param`	The parameter name followed by the description.	2
`@result`	The return value of the method.	1

Listing 2-24 Example of @method tag

```
/*!
 @method dateWithString:calendarFormat:
 @abstract Creates and returns a calendar date initialized with the date

 specified in the string description.
 @discussion [An extended description of the method...]
 @param description A string specifying the date.
 @param format Conversion specifiers similar to those used in strftime().
 @result Returns the newly initialized date object or nil on error.
*/
+ (id)dateWithString:(NSString *)description calendarFormat:(NSString
*)format;
```

For more usage examples, see the `ExampleHeaders` folder that accompanies the HeaderDoc distribution.

Using HeaderDoc

HeaderDoc includes two scripts, `headerDoc2HTML.pl`, which generates documentation for each header it encounters, and `gatherHeaderDoc.pl`, which finds these islands of documentation and assembles a master table of contents linking them together.

This chapter describes `headerDoc2HTML.pl`. For information on `gatherHeaderDoc`, see "Using gatherHeaderDoc" (page 37).

Running headerDoc2HTML.pl

Once you have a header containing HeaderDoc comments, you can run the `headerDoc2HTML.pl` 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 Command-line Switches

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

The `-C` switch causes HeaderDoc to output class contents as a composite page instead of breaking it up into separate pages for functions, data types, and so on.

The `-H` switch turns on inclusion of the `htmlHeader` line, as specified in the config file.

The `-O` switch enables "outer name only" type parsing, in which tag names for typedefs are not documented (for example, `foo` in `typedef struct foo {...} tdname;`).

The `-X` switch causes HeaderDoc to output XML content instead of HTML.

The -S switch causes HeaderDoc to include functions and data types from the superclass in the documentation of child classes (if they are processed at once).

The -b switch puts HeaderDoc into "basic" mode. In this mode, numbered lists are not automatically recognized, and embedded headerdoc comments are not removed from declarations.

The -d switch turns on extra debugging output.

The -h switch causes HeaderDoc to output an XML file containing metadata about the HeaderDoc output.

The -i switch tells HeaderDoc to output the body of macro declarations.

The -l switch tells HeaderDoc not to generate link requests in declarations.

The -m switch tells HeaderDoc to generate a man page for each function found in lieu of generating XML or HTML output.

The -o switch allows you to specify another directory for the output. For example:

```
> headerdoc2html -o /tmp MyHeader.h
```

The -q switch makes HeaderDoc operate silently:

The -s switch causes HeaderDoc to enter a comment stripping mode, in which it outputs a copy of your header file in the output directory from which all headerdoc comments have been removed.

The -t switch enables strict tagging mode, in which any function parameters not described with an @param tag result in a warning.

The -u switch disables sorting of functions, data types, and so on in the table of contents.

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 the Scripts Using MacPerl

Most of HeaderDoc runs on Mac OS 9 and earlier if MacPerl is installed. (You can get MacPerl from the CPAN ports page.) To run HeaderDoc using MacPerl:

- Change the line endings in the scripts and modules (*.pm files) from UNIX to Macintosh. Many text editors (BBEdit, for example) let you easily change line ending types.
- Run MacPerl, open headerDoc2HTML.pl and gatherHeaderDoc.pl and save them as droplets. You might save them with a different names (say, the script names minus the .pl extensions) to preserve the original versions.
- Now, you can drag a header file or folder of header files on each droplet in turn, and the files will be processed in place.

Note: Some advanced features, including automatic linking, man page output, and XML output will not work in Mac OS 9 because these require `libxml2`, which is only available for UNIX-based and UNIX-like systems.

Cocoa Front End

Kyle Hammond has made a Cocoa front end available for HeaderDoc. Mac OS X users can download this from their website at http://www.isd.net/dsl03002/CocoaProgramming.html.

Using gatherHeaderDoc

GatherHeaderDoc 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.

GatherHeaderDoc 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.

Running gatherHeaderDoc.pl

The `gatherHeaderDoc.pl` 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 "Configuring HeaderDoc" (page 51).

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 40).

`$$framework@@`

Inserts the framework name.

`$$frameworkdir@@`

Inserts the framework directory name (the name of the "`.hdoc`" file without the extension). This is useful when used with multiple landing page templates, as described in "Using Multiple Landing Page Templates" (page 40).

`$$frameworkdiscussion@@`

Inserts the framework discussion.

`$$frameworkabstract@@`

Inserts the framework abstract.

`$$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.

Using Multiple Landing Page Templates

HeaderDoc 8 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 add a line in your config file like this:

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

Next, create a pair of template files called `toctemplate.html` and `functions.tmpl`. In the file `toctemplate.html`, you can link to the functions index like this:

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

In the functions template, you can link to the main TOC like this:

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

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. Its contents are output in the location specified by the `masterTOCName` variable in the config file (`masterTOC.html` by default).

The second template file, `functions.tmpl`, results in a second HTML landing page whose name is derived from the directory name of the framework, followed by a dash, followed by the template filename (without any "`.html`" or "`.tmpl`" extensions), followed by "`.html`".

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 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 Description 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
<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
<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 1-1 (page 46) 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>Mac 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 1-2 (page 47) 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 1-3 (page 49) 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>
```

Configuring HeaderDoc

You can set values for some commonly altered variables. Currently, the configuration file lets you set these things:

copyrightOwner

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

defaultFrameName

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

compositePageName

The name of the file containing the printable HTML page (by default, `CompositePage.html`).

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.)

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 55) for more information.

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).

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 longer HTML headers. The contents of this file will be added to each content page if you specify the `-H` flag on the command line.

dateFormat

A string specifying the date format to be used by HeaderDoc. A few valid examples include `Y/M/D`, `M-D-Y`, `M/Y`, `Y`.

ignorePrefixes

Specifies a list of tokens to remove from HeaderDoc markup. Generally used to remove debug macros.

HeaderDoc Styles:

These contain CSS formatting for various parts of declarations. For example:
`funcNameStyle => background:#ffffff; color:#000000;`

commentStyle

style for comments

preprocessorStyle

style for preprocessor directives

funcNameStyle

style for function names

stringStyle

style for strings

charStyle

style for characters ('a')

numberStyle

style for numbers

keywordStyle

style for keywords

typeStyle

style for data types

paramStyle

style for function parameters

varStyle

style for variable names

useBreadCrumbs

Controls whether HeaderDoc will insert breadcrumb links into content pages instead of adding [Top] in the TOC. Valid Values are 0 or 1.

The path leading up to the current framework must be included manually in the `htmlHeader` or `htmlHeaderFile` directive. The breadcrumb is inserted wherever

```
<!-- begin breadcrumb --><!-- end breadcrumb -->
```

appears in the `htmlHeader` or `htmlHeaderFile` content.

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 38).

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

1. In the script itself (see the declaration of the `%config` hash near the top of `headerDoc2HTML`).
2. In the home directory of the user, in `Library/Preferences/com.apple.headerDoc2HTML.config`
3. In a file named `headerDoc2HTML.config` in the same folder as the script.

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
```

Configuration File Example

Listing 1-1 (page 53) is an example of a very basic HeaderDoc configuration file. Several additional examples are included as part of the HeaderDoc distribution.

Listing 6-1 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
```

Symbol Markers for HTML-Based Documentation

As HeaderDoc generates documentation for a set of header files, it injects named anchors () 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 forwars 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 A-1 (page 56).

Table A-1 HeaderDoc API reference language types

`c`	C
`occ`	Objective-C
`java`	Java
`javascript`	JavaScript
`cpp`	C++
`php`	PHP
`pascal`	Pascal
`perl`	perl script
`shell`	Bourne, Korn, Bourne Again, or C shell 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 A-2 (page 56).

Table A-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 or file-static data
`func`	function name (without '()')

Symbol Types for Languages With Classes

`cl`
: class name

`intf`
: interface or protocol name

`cat`
: category name, just for Objective-C

`intfm`
: method defined in an interface (or protocol)

`instm`
: an instance method 'clm' a class (or static [in java or c++]) method

C++ (cpp) Symbol Types

`tmplt`
: C++ class template

`ftmplt`
: C++ function template

`func`
: C++ scoped function (i.e. not extern 'C'); includes return type and signature.

Java (java) Symbol Types

`clconst`
: Java constant values defined inside a class

Note: The symbol value for method names includes the class name.

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 in NSWindow, the marker would be:

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

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)`.

HeaderDoc Class Hierarchy

```
HeaderElement (Root Class--any header entity that's significant)
  | (to HeaderDoc is a HeaderElement)
  |
  |
  |-----------APIOwner (Object that owns declared API)
  | |
  | |-------Header (Owner for header-wide API)
  | |
  | |-------CPPClass (Container for all non-Objective-C classes and
  | |                     C pseudoclass/COM Interface APIs).
  | |
  | |-------ObjCContainer
  |   |
  |   |-------ObjCClass (Owner for Objective-C class API)
  |   |
  |   |-------ObjCCategory (Owner for Objective-C category API)
  |   |
  |   |-------ObjCProtocol (Owner for Objective-C protocol API)
  |
  |
  |-----------Method (an Objective-C method)
  |
  |-----------Constant
  |
  |-----------Enum
  |
  |-----------Function (any non-objective-C function or method)
  |
  |-----------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.)
```

In addition to the classes shown above, the `headerDoc2HTML` script also uses the non–object-oriented modules `Utilities.pm`, `ClassArray.pm`, and `BlockParse.pm`. Most class instances are instantiated from `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, etc.

Finally, `gatherHeaderDoc` 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 Release Notes

HeaderDoc 8 is the latest incarnation of the HeaderDoc tool. It 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 nearly a rewrite of HeaderDoc from the ground up. It incorporates all of 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 62).

HeaderDoc 8 also adds a number of additional languages with various levels of support. These are described in "Languages Supported" (page 61).

Finally, HeaderDoc 8 adds a number of new (optional) tags for convenience. These are described in "New Tags" (page 63).

For additional information, see the documentation that is packaged with HeaderDoc.

Languages Supported

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

Table 7-1 HeaderDoc 8 Language Support

Language	HeaderDoc 7 support	HeaderDoc 8 support
C headers	yes	yes
C++	yes	yes
Objective C	yes	yes
C source code	no	yes

Language	HeaderDoc 7 support	HeaderDoc 8 support
K&R C sources	no	yes
Java	no	yes *
JavaScript	no	yes *
Pascal	no	yes
PHP	sort-of	yes **
Perl	no	yes **
Shell Scripts	no	yes **
Mach IPC Interface Defs	no	yes **

Note: * Java and JavaScript support only functions and classes.** 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
- Sopport for `//!` markup style for embedded HeaderDoc declarations
- Automatic linking of data types in declarations
- Improved C++ support (namespace/template/access)
- GatherHeaderdoc 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
- Beginnings of 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

(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.

- The GatherHeaderDoc default template is built-in. The format for this template is described in "Using gatherHeaderDoc" (page 37). Also see "Example gatherHeaderDoc Template" (page 41) for an example of the template format.