Version 1.8.4 Copyright ©1995 by Friday Software and Consulting, All Rights Reserved.

Autodoc - Release 1.8.4

Guide for Use with Objective-C Source Code

Adam Swift, July 1995

Foreword

Autodoc is a utility that generates NEXTSTEP-style Rich Text Formatted documentation for Objective-C classes, categories, and protocols by extracting comments imbedded in source code files. Autodoc will also produce documentation for functions, definitions, macros, typedef's and global variables.

Autodoc was developed by Adam Swift <aswift@friday.com>. It was initially funded by Information Technology Solutions and has been donated free of charge to the Public Domain. Currently, Friday Software and Consulting is the entity maintaining Autodoc. As author of Autodoc, I would like to express my thanks to ITS for allowing and encouraging me to make Autodoc freely available to the public. It is important to see the chapter: *Utterly Fun Legal & Warrantee Information* before using Autodoc.

Thanks also to Bill Bumgarner <bbum@friday.com>, Todd Anthony Nathan <todd@icebox.com>, Kim Shrier <kim@media.com>, and Craig Kelley <ckelley@capaccess.org> for their bug fixes and improvements. Contributions, suggestions and questions regarding Autodoc are *very* welcome, please send them either directly to

Adam Swift or to the Autodoc mailing list **<autodoc@friday.com>**. To subscribe to the mailing list, send a message with the subject, "subscribe", to <autodoc-request@friday.com>.

Table of Contents

1. Installing Autodoc

This chapter describes how to install Autodoc and tries to show you where to look if things go wrong. Note that there is an INSTALL file included in the release which gives a quick summary of the installation procedure.

The Autodoc utility is actually a Perl script, in other words, you need to have access to the Perl program in order to use Autodoc. If you do not already have Perl, it is generally available at: ftp.uu.net. Autodoc was written to work with Perl version 5.001, but should work with version 5.000

Autodoc must be able to access the perl program in order to execute the script. Autodoc looks for the perl program as `perl5` in `/usr/local/bin`, if you cannot put perl there, you have to edit the first line of Autodoc to refer to the actual location of perl. (e.g. If the Perl program is actually in /LocalDeveloper/Unix/bin, change the first line from: "`#!/usr/local/bin/perl5`" to: "`#!/LocalDeveloper/Unix/bin/perl`".)

Autodoc must be able to access certain perl modules in order to run, the modules which were written specifically to support Autodoc (and therefore are freely available) are

available from the author <aswift@friday.com>, and the common modules available at most perl resource ftp site. The complete list of required modules follows:

flush.pl - *common*
Getopt/Long.pm - *common*
Cwd.pm - *common*
Exporter.pm - *common*
AutodocSupport/FileSupport.pm
AutodocSupport/LogDebug.pm
AutodocSupport/ScanFile.pm
AutodocSupport/ParseComments.pm
AutodocSupport/GenerateRTF.pm
AutodocSupport/ReadSource.pm
AutodocSupport/DumpDocs.pm

The custom autodoc modules should be installed alongside to the perl5 modules on your filesystem (eg. `/usr/local/lib/perl5/`). If you cannot copy the modules there, or if autodoc cannot locate the custom perl modules, you can instruct autodoc to find them in any directory you want. To do so, set the environment variable `AD_PMLIBDIR` to point to the directory which all of the .pm files are installed in:

For example, to include files from /Users/username/lib/perl5, add the following line to your .cshrc file:

```
setenv AD_PMLIBDIR '/Users/username/lib/perl5>'
```

Autodoc is a command line utility, so the easiest way to run it is from a login shell. In the shell, change directories to the directory Autodoc is in, and type "`./autodoc`" at the prompt to see if Autodoc can run as it is configured on your computer. (You must set the executable flag on Autodoc file permissions in order to run it by just typing "`./autodoc`") If everything works properly, Autodoc should list its Invocation Options (See *Invocation Options*

and Miscellaneous.)

If you get an error, or nothing happens, it is most likely a problem interacting with Perl. To see if it is working and accessible, try executing perl manually with the entire file path prepended, and the -v option to display the version.

2. Generating Documentation from Source Files

This chapter describes the kinds of source code you can document, and how to produce formatted documentation from source code files.

2.1 Types of Source Code which Autodoc can Document

Autodoc can document Objective C classes, categories, and protocols. Also, it can document header files which define collections of resources (such as defines, macros, typedefs, functions, and global variables.) The following table shows the source files which Autodoc must use in order to document these different types of source code.

class	**.h** file with class interface **.m** file with class implementation
category	**.h** file with category interface **.m** file with category implementation
protocol	**.h** file with protocol declaration **.mdoc** file with protocol "implementation"*

* NOTE, Protocols do not have any implementation in Objective C; however, in order to document a protocol, Autodoc looks for a file with the same root name as the '.h' file, but with

a '.mdoc' suffix. The mechanism for documenting protocols parallels documenting classes and categories.

For Autodoc to include a "Protocol Description" in the documentation it produces, you should include the line '@implementation ProtocolName' immediately before the protocol documentation. Each method in the protocol can be individually documented by putting an autodoc comment after that method line.

2.2 Generating Documentation for a Single Class, Category, or Protocol

The default operating mode of Autodoc is to read each of the class, category, or protocol names supplied on the command line, extract comments from the named source files (by scanning the files which have the same root name as the class or category name, with ".h" and ".m" appended, for protocols, ".h" and ".mdoc") and produce Rich Text Formatted document files for each (by adding ".rtf" to the root file name). The files which Autodoc produces by default are saved to the directory where the source files are located.

For example, suppose I have the source files `MyClass.h` and `MyClass.m`, and they are located in directory `/Users/me/MyProject`; I could generate documentation for them by typing the following into a shell:

```
% autodoc /Users/me/MyProject/MyClass
autodoc producing /Users/me/MyProject/MyClass.rtf ... done
%
```

And in the directory `/Users/me/MyProject` there should be a file named `MyClass.rtf` which contains the formatted documentation for MyClass. (Note, I must have write permissions for the `/Users/me/MyProject` directory for this to work.)

2.3 Generating Documentation for a Project's Source Files

In order to produce documentation for a project directory full of source files (plus recursively descending subdirectories of source code,) then to output all of the documentation in a separate destination directory, you need to supply a set of command line options to Autodoc (See *Invocation Options and Miscellaneous*.)

For example, suppose I have a project filled with source files and subprojects based in directory `/Users/me/MyProject`, and I want to produce the documentation files in `~/Docs/MyProject`; I could generate documentation for them by typing the following into a shell:

```
% autodoc -tree -proj /Users/me/MyProject -dest ~/Docs/MyProject
autodoc creating ~/Docs/MyProject ... done
autodoc producing ~/Docs/MyProject/MyClass.rtf ... done
autodoc producing ~/Docs/MyProject/(other source files)
autodoc creating ~/Docs/MyProject/MySubproject ... done
autodoc producing ~/Docs/MyProject/MySubproject/MyOtherOb
ject.rtf ... done
...
%
```

And in the directory `~/Docs/MyProject` there should be a file named `MyClass.rtf` which contains the formatted documentation for MyClass. (Note, I must have write permissions for the directory `~/Docs/MyProject`, or if it does not already exists, the directories which need to be created.)

3. Writing Compliant Source Code and Comments

This chapter outlines the information that you need to provide in your source code in

order to produce documentation files which look like the NEXTSTEP developer documentation.

The information which is extracted from the source files fits into two categories:

- Elements from the actual definitions of the class, category, or protocol; and
- Specially marked comments which are interpreted according to their location in the source.

The comments which are used in producing the documentation must appear within `/*"` and `"*/` comment delimiters. This style of comment is referred to as *Autodoc style comments*. The one exception to this convention is copyright information, since it is automatically generated by DevMan, that information appears inside regular c style comment delimiters: `/*` and `*/`.

In all of the examples given below, when text is shown in bold and italics, that text is what will be extracted by Autodoc.

3.1 Copyright Info

The version number and year are extracted from the $Id: line inside regular (/* ... */) comments in the header file (the $Id: line is maintained by DevMan.app.) *Note that the copyright owner may only be specified by using the command line flag -c.*

Example: `$Id: MyClass.h,v` ***1.1 1994***`/12/06 ...`
Fail Case: If @interface is found before $Id:, the default is used.
Default: The version is not set, the year is set to the current year.

3.2 Class Name

Extracted from the class @interface declaration in the header file.

Example: `@interface MyClass : SuperClass`

Fail Case: If @interface or @protocol is not found in the header file, the source is treated as a "Resource Collection" not related to any class, category, or protocol.

3.3 Category Name

Extracted from the category @interface declaration in the header file.

Example: `@interface MyClass (CategoryName)`

Fail Case: If @interface or @protocol is not found in the header file, the source is treated as a "Resource Collection" not related to any class, category, or protocol.

3.4 Protocol Name

Extracted from the protocol @protocol declaration in the header file.

Example: `@protocol MyProtocol`

Fail Case: If @interface or @protocol is not found in the header file, the source is treated as a "Resource Collection" not related to any class, category, or protocol.

3.5 Inherits From

Extracted from the class @interface declaration in the header file. Optional Autodoc style comments give further inheritance information.

Example: `@interface MyClass : SuperClass /*" SuperSuperClass : ... : RootClass "*/`

Fail Case: If @interface or @protocol is not found in the header file, the source is treated as a "Resource Collection" not related to any class, category, or protocol.

3.6 Conforms To

Extracted from the class, category or protocol header file, after the @interface or

@protocol declaration.

Example: `@interface MyClass : SuperClass <`***`MyProtocol, MyOtherProtocol`***`>`

Fail Case: If @interface or @protocol is not found in the header file, the source is treated as a "Resource Collection" not related to any class, category, or protocol.

3.7 Declared In

Extracted from the header file inclusion (#include or #import) in the source file.

Example: `#import <`***`filepath/MyClass.h`***`>` (Note, the `<x>` may also appear as `"x"`)

Fail Case: If @implementation is found before the include/import, the default is used.

Default: The header file name with no file path, eg "MyClass.h"

3.8 Class, Category, or Protocol Description

Extracted from Autodoc style comments immediately following the class, category, or protocol @implementation in the source file. Empty lines (two or more newlines) in the comment will generate separate paragraphs in the produced documentation.

Example:
```
@implementation MyClass
/*" MyClass is .... "*/
```

Fail Case: If text other than the comment is found after the @implementation, the default is used.

Default: "No class description."

3.9 Instance Variables

Extracted from instance variable declarations in the header file. Optional Autodoc style comments which follow a variable are associated with it.

Example: `NXRect *myBounds; /*" The size and location of MyClass "*/`

Fail Case: If methods follow the @interface before variables, this section in not included in the documentation.

3.10 Method Types

Extracted from method declarations in the header file. Optional Autodoc style comments which precede methods define a method block label in the documentation.

Example:
```
/*" Initializing a MyClass "*/
- init;
```

Fail Case: If @end is found before methods, this section in not included in the documentation.

3.11 Method Description

Class and instance method descriptions are extracted from Autodoc style comments in method definitions in the source file. The comments must appear after the method name, but before the open bracket. Note that argument variables (a method's arguments) are automatically set in italicized text.

Example:
```
- init
/*" Initializes and returns ... "*/
```

Fail Case: If the "{" is found before the method description, the default is used.

Default: "No method description."

3.12 Functions

Extracted from function declarations in the header file. Function's are only included in the documentation if they appear *before* the @interface line in the header file, and they must begin with the "extern" modifier. Documentation should appear before the function declaration and functions will be grouped by documentation.

Example:
```
/*" Function documentation for foo() and bar() "*/
extern void foo();
extern int bar();
```

Fail Case: If @interface is found before functions, this section in not included in the documentation.

3.13 Macros

Extracted from macro definitions in the header file. Macro's are only included in the documentation if they appear *before* the @interface line in the header file. Documentation should appear before the macro definition and macros will be grouped by documentation.

Example:
```
/*" Macro documentation for FOO() and BAR() "*/
#define FOO(x) whatever(x)
#define BAR(x) whateverelse(x)
```

Fail Case: If @interface is found before macro's, this section in not included in the documentation.

3.14 Symbolic Constants

Extracted from definitions in the header file. Constant's are only included in the documentation if they appear *before* the @interface line in the header file. Documentation should appear before the constant definition and constants will be grouped by documentation.

Example:
```
/*" Constant documentation for BAZ "*/
#define BAZ    29
```

Fail Case: If @interface is found before constants, this section in not included in the documentation.

3.15 Defined Types

Extracted from definitions in the header file. Defined types are only included in the documentation if they appear *before* the @interface line in the header file. Documentation should appear before the typedef definition and defined types will be grouped by

documentation.

Example:
```
/*" Defined type documentation for FOOType "*/
typedef void   FOOType;
```

Fail Case: If @interface is found before defined types, this section in not included in the documentation.

3.16 Global Variables

Extracted from declarations in the header file. Globals are only included in the documentation if they appear *before* the @interface line in the header file, and they must begin with an "extern" modifier. Documentation should appear before the global declaration and globals will be grouped by documentation.

Example:
```
/*" Global variable documentation for RealGlobal "*/
extern int     RealGlobal;
```

Fail Case: If @interface is found before global variables are defined, this section in not included in the documentation.

4. Source Code Commenting Style and Philosophy

This chapter describes the style and philosophy followed in the documentation of source code in the NEXTSTEP Developer Documentation. In order to produce documents of the same quality level, it is important that the text content be as consistent as the layout and formatting.

The best model to use for class, category and protocol design and documentation is the NEXTSTEP Developer Documentation.

4.1 Text Formatting Switches for Autodoc Style Comments

Autodoc provides switches which set the text appearing inside the documentation file into a special font style. These font style modifiers can be used inside any Autodoc style comment.

`#word *and* #{any text}`

Used to set the word(s) into a **bold** type face. Note that the colons will not terminate the word in the example: `#aMethod:withThree:arguments:.`

`%word *and* %{any text}`

Used to set the word(s) into an *italic* type face.

`_{item description...}`

Used to set the item and the following words in the description onto a line by itself, seperated by a tab. Use this to create a line in a list of items and descriptions.

NOTE: In order to have % and # characters appear in the documentation as % and # characters, you should put the escape character \ immediately before them, as in: '\#' and '\%'.

4.2 Text Formatting Guidelines

All text formatting is automatically configured by Autodoc, except for special style changes within the descriptions for the class, instance variables, methods, and method blocks. Each of these descriptions are placed inside Autodoc style comments in the source code.

The only automatic formatting are: methods argument names will be automatically italicized within a method description, and the words **self, id**, **super** and **nil** will always be bolded.

WHEN TO USE BOLD TEXT

When methods are referred to in descriptions, they should appear in bold type, and the arguments and their types should not be included (e.g. **renewRows:cols:**.)

WHEN TO USE ITALIC TEXT

When you wish to emphasize an important concept, set it in italics (e.g. Each Window provides a *field editor*.) Italics should not be used for instance variables, and in general, instance variables should not be referenced directly in the documentation. Note that argument variables (a method's arguments) are automatically set in italicized text.

5. Invocation Options and Miscellaneous

This chapter describes the command-line switches (options) which may be used when executing Autodoc. In addition to these command line switches, the `AD_COPYRIGHT` environment variable can be used to specify a default copyright owner. Command line switches can be used in full, or as single letters where unique (eg. `-project` can be specified as `-p`)

`-project` *projectdirectory*

Used to specify a project directory for reading in source files. If no source files are specified, then all files in *projectdirectory* (and its subdirectories) which form a pair, "FileName.h" and "FileName.m" (or "FileName.mdoc" for protocols), as well as all files ending in ".h" (unless the `-nosingles` switch is specified,) are used to produce documentation. The documentation produced will be saved under the root name for each of the file pairs, and will be stored in the source files' directory (unless the `-destination` option is used.)

`-destination` *destinationdirectory*

Used to specify a destination directory for the documentation files produced. If the `-tree` option is also used, any source files which are prepended with a file path, will be placed in a subdirectory of the destination directory with the same file path.

`-copyright` *copyrightowner*

Used to specify an copyright owner for the copyright line on top of the documentation file. This will override the copyright owner specified by the environment variable `AD_COPYRIGHT` if both are used.

`-version`

Displays the version number and copyright information for Autodoc.

`-help`

Displays the command line options available in Autodoc, with synopses of their purpose.

`-timestamp`

Includes the current time and date in the copyright line of the documentation files created.

`-nosingles`

Excludes unpaired source files from the files used to produce documentation (i.e. if the file "FileName.h" is in a directory which does not contain "FileName.m", no documentation will be generated from it.)

`-rtf`

When combined with the `-project` and `-destination` options and no source files are explicitly specified, this will copy all files found with an 'rtf' or 'rtfd' extension found in the project directory or its subdirectories to the destination directory.

`-force`

Force autodoc to generate documentation files, even if the documentation files already

exist and are newer than the source files. By defualt, autodoc will not overwrite documentation files which are newer than their source files.

`-tree`

Specifies that the documentation should be generated into a directory tree based on the directories traversed in locating the source files. This can be combined with the `-project` and `-destination` options to produce subprojects source file documentation in subdirectories of the destination directory.

`-silent`

Makes Autodoc operate in silent mode, the only messages printed are error messages logged to stderr. Normally Autodoc alerts the user whenever a file or directory is created.

`-Debug` *level*

Makes Autodoc operate in DEBUG mode, verbose debugging messages are logged to stderr. The debugging levels for Autodoc range fall in the the following categories:

1 Source code files being processed and general Autodoc configuration information
2 State of processing source files as they are processed
3 Source code data that is located, and a summary of what was found
4 Detailed information about all source data that is extracted
5 Complete logging of every line of text read in from the source files.

6. Utterly Fun Legal &Warranty Information

Because the Autodoc is licensed free of charge, there is no warranty for the program. Copyright holder, Information Technology Solutions, is providing this program "as is" and this program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR

PURPOSE.