Autoduck User's Guide

To display a list of topics by category, click any of the contents entries below. To display an alphabetical list of topics, choose the Index button.

Basics

These topics describe the basic approach of Autoduck:

Tag Reference

The following topics describe the tagset defined in AUTODUCK.FMT:

Using Autoduck.Exe

These topics describe how to use the application, and how to structure your makefile entries:

Customizing Output

Autoduck lets you change the format of output produced by Autoduck. You can add tags, change the structure of topic indexes, and even define new output file types.

Format-File Reference

Tags and output strings are defined in a format file like AUTODUCK.FMT. You can customize AUTODUCK.FMT or define your own format file.

[INDEX]
[PARAGRAPH]
[TEXT]
[TOKEN]
[TOPIC]

About Autoduck

About Autoduck

The sources for this Help file were generated by Autoduck, the source code documentation tool that generates Print or Help files from tagged comments in C, C++, Assembly, and Basic source files.

For more information, contact Eric Artzt (erica@microsoft.com).

About Autoduck

Autoduck is a command-line utility that extracts specially tagged comment blocks from programming source files and generates rich text files containing the contents of those comment blocks. Autoduck has traditionally been used to document programming APIs.

Placing API documentation within the source files helps programmers disseminate information about a developing codebase. Autoduck can generate online Help files containing full hypertext coding with links and keyword lists. Typically, Autoduck is integrated into the build process, so a new Help database can be automatically generated each build.

Integration of documentation with code makes it easier to keep the documentation up to date. When developers make changes to APIs, they can quickly update the comment blocks at the same time. When APIs are released for use by outside customers, User Education personnel can edit the comment blocks, add example code, and generate final RTF files for inclusion in printed or online documentation.

Autoduck Comment Blocks

AUTODUCK.EXE scans through a source file and extracts information marked with Autoduck tags. Autoduck information is stored in *topics*, discrete units of information. For example, a topic might consist of a function reference description or a discussion of sample code. The following example shows an Autoduck topic:

```
// CLazyInterface::QueryInterface
//
// @mfunc  Implements <om IUnknown.QueryInterface>.
//
// @commThe default implementation delegates to the
//              controlling unknown.
//
HRESULT FAR PASCAL CLazyInterface::QueryInterface(

REFIID riid,        // @parm Requested interface.
LPVOID FAR *ppv)    // @parm Where to store the returned <f AddRef>'d
               //          interface pointer.
{
 DPF3("CLazyInterface::QI('%s')\n", DebugIIDName(3, riid));

 // delegate to controlling unknown
 return m_punkOuter->QueryInterface(riid, ppv);
}
```

The @mfunc tag marks the start of a new topic describing a C++ member function. The @comm and @parm tags identify paragraph types within the comment blocks (these mark comment and parameter descriptions, respectively). The angle bracket codes (<f > and <om >) are text tags marking special types of text (functions and object methods).

Autoduck is Free!

Autoduck is free. There is no licensing fee or restriction. The application is considered sample code and is not supported by Microsoft Corporation. Here's the legalese:

THIS TOOL IS NOT SUPPORTED BY MICROSOFT CORPORATION. IT IS PROVIDED "AS IS" BECAUSE WE BELIEVE IT MAY BE USEFUL TO YOU. WE REGRET THAT MICROSOFT IS UNABLE TO SUPPORT OR ASSIST YOU SHOULD YOU HAVE PROBLEMS USING THIS TOOL.

You are free to use the application, distribute it to others, and/or modify the source code. If you make changes, have comments, or find bugs, you can mail them to me, Eric Artzt, at my Internet address:

erica@microsoft.com

I can't guarantee that I will fix your bug, or even answer your question right away, because regrettably, supporting Autoduck is not what I do for a living - rather, I produce children's software products - but I will do what I can as time (and interest) permits.

Features of Autoduck

The following are some interesting features of Autoduck:

Flexible Tag Definition

Autoduck tags are defined in a text file called a formatting file. The formatting file defines which tags are recognized as well as the RTF output for a tag. The formatting file makes it easy to define your own tags or modify the formatting applied to topic text. Autoduck comes with a standard formatting file, AUTODUCK.FMT, that defines commonly used tags for C, C++, and the OLE2 Component Object Model (COM).

For more information, see “Defining Tags and RTF Output Strings”.

Source Parsing

Autoduck can extract certain information from the C/C++ source declarations. For example, developers can type a function name in the comment block, or allow Autoduck to extract a function name from the function header.

For more information, see “Source Parsing”.

Extraction Tags

When performing an Autoduck build, you can use extraction tags to specify which Autoduck topics are included. This is useful when your codebase has both internal and external APIs, or when you want to generate RTF files for a subset of topics.

For more information, see “Conditional Topic and Paragraph Extraction”.

Topic Logs

Autoduck can reference a topic log file when generating hypertext links for RTF. A topic log file lists the set of topics available to link to. If a topic is listed in the topic log, Autoduck can create a hypertext link to it. If the topic is not listed, Autoduck can generate alternate formatting (like bold text).

For more information, see “Topic Logs”.

Autoduck Tags

Autoduck uses tags to identify what type of information is contained in a comment block. For example, a tag might identify a paragraph as a description of a parameter or return value of a function.

An Autoduck tag consists of an ampersand (@) followed by a tag name. Most Autoduck tags are defined in a format-information file, which is required by the AUTODUCK.EXE tool. The format-information file defines the number of fields within a tag, the formatting strings to output for the tag, and extraction information for the tag. AUTODUCK.EXE only recognizes tags that appear as the first item within a line of comment text.

There are three basic types of Autoduck tags:

- The @DOC tag, which signals the beginning of Autoduck information within a source file and defines flags used to determine which Autoduck information to extract.
- Topic tags that define new Autoduck topics.
- Paragraph tags that define new types of paragraphs within Autoduck topics.

All three types of tags must conform to the following formatting conventions:

- Tag names must begin with the "@" character.
- White space is not allowed within a tag name.
- Tag names are not case-sensitive.

Topic and paragraph tags contain one or more *fields* of text. Fields are delimited by pipe (|) characters. A field can contain multiple paragraphs of text (paragraphs are delimited by consecutive newlines). For example, the following @PARM tag defines information for a paragraph about a function parameter:

```
//@parm int | iType | Specifies the type.
```

Autoduck output combines formatting codes stored in the formatting specification file with field text parsed from the Autoduck comments in the source file. In the above example, the parameter name *iType* might be output in italics, followed by an indented paragraph containing the third (description) field.

In addition, Autoduck provides for text tags that identify special types of paragraph text (for example, a function or message name). The three types of @ tags, as well as the text tags, are described in the following sections.

@DOC Tag

The @doc tag must be the first Autoduck tag encountered in the source file. The @doc signals the beginning of Autoduck information within a source file and identifies *extraction flags*, tokens used to classify Autoduck topics. For example, you can classify Autoduck information as EXTERNAL or INTERNAL, then extract only those topics falling under the

EXTERNAL category.

For more information on @DOC, see “Conditional Topic and Paragraph Extraction”.

Topic Tags

An Autoduck topic begins with a topic tag. Topic tags are defined in the [TOPIC] section of the formatting file.

The following example shows a @FUNC tag, which defines a new topic describing a function:

```
//@func int | MyFunction | This function performs a useful task.
```

The function has three fields, for the return value, function name, and function description. The text in these fields is written to the output file.

Paragraph Tags

A paragraph tag defines a paragraph within a topic. Paragraph tags are defined in the [PARAGRAPH] section of the formatting file.

For example, the following @PARM tag defines a topic paragraph containing a description of a function parameter:

```
//@parm char *| szText | Specifies a pointer to a text string.
```

Text Tags

Text tags can be used within topic and paragraph tags to identify references to document elements and to generate special characters such as the trademark symbol. Text tags conform to the following guidelines:

- The tag and its fields are enclosed in angle brackets (< >).
- Fields are separated by a period (.) or double colon (::).
- Tag names are not case-sensitive.

A text tag begins with the opening bracket, followed immediately by the tag name. Fields, if present, are placed following a single space following the tag name:

<tagname field1.field2.field3>

Text tags are defined in the [TEXT] section of the formatting file.

For example, the following @FUNC tag contains the text "YourFunction" marked with a function name type:

```
// @func int | MyFunction | This function performs a useful task. But
// always be sure to call <f YourFunction> first!
```

Autoduck Comment Blocks

Autoduck-tagged text can reside in any text file, as long as it resides within a C/C++, Assembler, or BASIC comment block.

Note the following guidelines for Autoduck comment blocks:

- Autoduck topics can reside in a single comment block, or they can span multiple comment

blocks.

- Autoduck comment blocks can begin anywhere on a line (they can be preceded by source statements).
- C-language comment blocks can use the slash/asterisk format (/* closed by */) or the slash/slash format (//).
- Assembly-language comment blocks must be a series of comment statements beginning with semicolon (;) characters.
- Basic-language comment blocks must be a series of comment statements beginning with apostrophe (') characters.
- A @DOC tag must precede any autoduck tags within the source file. The extraction flags established by an @doc tag remain in effect until the end of the file, or until the next @doc tag encountered.

Examples of Comment Blocks

This section contains several examples of Autoduck blocks.

The following comment block includes an @DOC tag and @FUNC tag:

```
/*
  * This text is ignored by Autoduck.
  *
  * @doc
  *
  * @func int | MyFunc | This function performs a useful task.
  */
```

Following are other variations, using different types of comment delimiters:

```
// C++ slash-slash comment
//
// This text is ignored by Autoduck.
//
// @doc
//
// @func int | MyFunc | This function performs a useful task.

; Assembly language comment
;
; This text is ignored by Autoduck.
;
; @doc
;
; @func int | MyFunc | This function performs a useful task.

' BASIC comment
'
' This text is ignored by Autoduck.
'
' @doc
'
' @func int | MyFunc | This function performs a useful task.
```

Noise Characters

The following characters are considered to be noise characters and are stripped from text

before it is output:

- Leading white space characters (spaces and tabs).
- Asterisks, semicolons, and apostrophes in the first character position, and any similar characters immediately following the first character position.
- Asterisks in the second character position if the first character is a space.

Topics Spanning Multiple Comment Blocks

Autoduck blocks can span several comment blocks. The Autoduck-tagged text is appended to the preceding Autoduck topic within the source file.

The following example includes three separate comment blocks. The function topic is started in the first comment block, and the parameter paragraphs are specified in later comment blocks:

```
// SetEmptyFields
//
// @func This function sets all empty fields to point to stub text.
//
void SetEmptyFields(
 PTAG ptag,   //@parm Specifies the tag to fill.
 int nFields)  //@parm Specifies how many fields.
{
 int i;

 for(i = ptag->nNumFields; i < nFields; i++)
     ptag->szField[i] = gszEmptyField;

 ptag->nNumFields = nFields;
}
```

This example also shows the source-parsing capability of Autoduck; normally, the @FUNC tag requires three fields (return value, function name, and description), so Autoduck parses the function header immediately following the first comment block to obtain the return value and function name.

Source Parsing

Autoduck has the capability of extracting tag fields from C/C++ and Visual Basic source statements. The purpose of this source-parsing capability is to eliminate redundant entry of type and variable declarations. To enable source parsing for an autoduck tag, you must add a ".PARSESOURCE" entry in the formatting file specification for that tag.

If the required source-parsing entry is specified in the formatting file, Autoduck attempts to parse the tag fields from the source text if the required fields are not present in the tag itself. For example, if the @parm tag expects three fields (parameter type, name, and description), and only one tag (description) is present, Autoduck will check the formatting-file entry to see whether source parsing is enabled. If it is, Autoduck will try to extract the missing fields from the source text.

Supported Source Parsing Configurations

Autoduck can parse source text from two locations:

- A source declaration following an Autoduck comment block

- A source declaration occurring on the same line on which an Autoduck comment block begins (comment following the source element)

Here are examples of both types:

```
//@func This is my function.
//@parm This is a string parameter.
//@parm This is a integer parameter.

int MyFunction(char *sz, int i)
{
...
}

//@func This is another function.

int AnotherFunction(
 char *sz, //@parm This is a string parameter.
 int i)    //@parm This is a integer parameter.
{
...
}
```

Parameter and Structure Field Parsing

Autoduck can parse the type specifier and variable name from a function parameter or structure field. The parameter/field type and name are deposited in the first two fields of the tag record.

To enable parameter or field parsing, you must add either the statement ".PARSESOURCE=parameter" or ".PARSESOURCE=field" to the tag definition.

Function and Member Function Parsing

Autoduck can parse the return type, function name, and (if applicable) class name from a function or member function definition.

For functions, the return type and function name are deposited in the first two fields of the tag structure. For member functions, the return type, class name, and function name are deposited in the first three fields of the tag structure.

To enable function or member function parsing, you must add either the statement ".PARSESOURCE=function" or ".PARSESOURCE=memberfunction" to the tag definition.

Class Parsing

Autoduck can parse the class name from a class declaration. It does not parse a "const" keyword; to add a "const" keyword, use an "@this const" tag within the comment block.

To enable class parsing, you must add the statement ".PARSESOURCE=class" to the tag definition.

Enumeration Member Parsing

Autoduck can parse the names of enumeration members.

To enable enumeration member parsing, you must add the statement ".PARSESOURCE=emem" to the tag definition.

Nesting Topics Inside Topics

To document inline member functions, class structures, and class enumeration types, you can created nested topics that generate both an Autoduck paragraph and a separate Autoduck topic.

For example, consider the following C++ class declaration:

```
class CString {

public:
 CNested(void) { m_szText = NULL; }
 ~CNested(void) { Reset(); }

 void Reset(void) { if(m_szText) delete m_szText; m_szText = NULL; }

 enum CompareFlags {
    compNormal,
    compIgnoreCase,
    compFuzzy,
 }

 int Compare(const char *szCompText, int nCompFlags = 0);

 void Set(const char *szText);

private:
 char *m_szText;
};
```

The inline constructor and member functions can have their own topics, as can the CompareFlags enumeration type.

Tagging Nested Topics

To generate a topic for nested constructs, tag the construct with a paragraph tag, but add a topic tag after the paragraph tag.

Also, you can define paragraph tags that only apply to the nested topic, and are not picked up as part of the main topic.

For example, here is the same class Autoduck'ed:

```
//@class A simple string class.

class CString {

public:
 //@cmember,mfunc Constructor, initializes string to empty.

 CNested(void) { m_szText = NULL; }

 //@cmember,mfunc Destroys the string, if present.

 ~CNested(void) { Reset(); }

 //@cmember,mfunc Destroys the string, if present.

 void Reset(void) { if(m_szText) delete m_szText; m_szText = NULL; }
```

```
//@cmember,menum Comparison flags for the <mf .Compare> function.

enum CompareFlags {
    compNormal,             //@@emem Case-sensitive compare.
    compIgnoreCase,         //@@emem Case-insensitive compare.
    compFuzzy,              //@@emem Fuzzy compare, if words sound
}                            // alike

//@cmember Comparison function.

int Compare(const char *szCompText, int nCompFlags = 0);

//@cmember Sets the text of the string.

void Set(const char *szText);

private:
//@cmember Pointer to string text.

char *m_szText;
};
```

Format-File Entries for Nested Topics

To generate the topic tag for a nested topic, Autoduck first parses the paragraph tag, then copies the paragraph fields over into the topic tag. The **.MAP** format-file entry defines how paragraph-tag fields are mapped over to the topic tag.

For example, the following entry for the **@cmember** tag defines how fields in that tag are mapped over to topic fields. There are **.MAP** entries for the **@mfunc**, **@menum**, and **@mstruct** tags.

```
.tag=cmember, help, 4, 2
.pre=$[classhdr]
.format=$(reset)$(term1){\uldb $1}{\v #1} {\uldb $2}{\v #class.1__#2}\par
$(reset)$(def1)$4\par

.if=exists($class.1::$<2),fieldempty(3),exists($1)
.parsesource=classmember
.map=mfunc,$1,$t.1,$2,$4
.map=enum,$1,$2,$4
.map=struct,$1,$2,$4
```

The **@mfunc** entry maps four **@cmember** fields over to the topic tag. The first **@mfunc** field receives the first **@cmember** field; the second **@mfunc** field receives the first field of the enclosing topic (**@class**) tag; and the third and fourth fields receive the second and fourth fields, respectively, of the **@cmember** tag.

Output Types

Through the format (.FMT) files provided with the package, Autoduck currently supports three types of output:

- Rich Text Format (RTF) for Print: RTF is the interchange format used by Microsoft word processors. Many other word processors support the RTF file format. Autoduck can create .RTF output for print documents (i.e., a document designed to be opened in a word

processor and printed).

Use the /RD option to generate RTF Print output. This is the default output format if none is specified.

Print RTF definitions are included in AUTODUCK.FMT, which is used by default.

- Rich Text Format (RTF) for Windows Help: RTF is also used as the input format for Microsoft Windows Help, a hypertext/help application available on Microsoft Windows. Autoduck outputs a different flavor of RTF for use in Help titles - in this case, the output includes Help compiler directives coded as footnotes in the RTF. Autoduck also creates a Help Project File (.HPJ file) needed by the Help compiler.

  Use the /RH option to generate RTF Help output.

  Help RTF definitions are included in AUTODUCK.FMT, which is used by default.

- HTML: The newest output type supported.

  Use the /RHTML flag to generate RTF Help output. You must also specify the HTML.FMT formatting file (/f HTML.FMT).

  HTML definitions are included in HTML.FMT.

Example

Use the following command line to output Help RTF for all the .CPP files in the current directory:

```
autoduck /rh /o myproject.rtf *.cpp
```

Use the next command line to output print RTF:

```
autoduck /o myproject.rtf *.cpp
```

Use the next command line to output HTML:

```
autoduck /rhtml /f c:\autoduck\html.fmt /o myproject.htm *.cpp
```

Using AUTODUCK.EXE

AUTODUCK.EXE is a console application that extracts and formats Autoduck source files.

The following is the command-line syntax for AUTODUCK.EXE:

```
AUTODUCK [/v] [/e] [/n] [/a] [/u] [/r[dh]] [/t[0-9]]
    [/o filename] [/l filename]
    [/f filename] [/c filename]
    [[/x id]...] [[/d name=text]...] files
```

Option	Description
[/v]	Prints detailed status information to the console.
[/e]	Suppresses warnings about empty fields.
[/n]	Suppresses topic and .HPJ output; only creates log file (if specified).
[/a]	Appends RTF and log-file output to existing files.
[/u]	Suppresses sorting of topics.
[/rd]	Generate RTF for Print, using the formatting information

	tagged either as "DOC" or "BOTH". This is the default.
[/rh]	Generate RTF for Help, using the formatting information tagged either as "HELP" or "BOTH".
[/t[0-9]]	Sets the tab size for example tags. Use the same setting used in your text editor. The default value is 8.
[/o filename]	Use output file <filename>. If no output file is specified, Autoduck creates an output file with the same filename as the first input file, and extension .RTF.
[/l filename]	Creates topic log <filename> using the topics extracted in the current build. The topic log is a list of topic names included in the current build.
[/f filename]	Use format file <filename>. If no format file is specified, Autoduck searches for Autoduck.FMT in the directory where AUTODUCK.EXE is stored, and then in the directories referenced by the PATH environment variable.
[/s filename]	Use supplemental format file <filename>. You can specify a supplemental format file in addition to the main format file. Entries in the supplemental format file override or add to the entries in the main format file. Using a supplemental file, you can define project- or group-specific variations to the default AUTODUCK.FMT file. Note that you can also insert your additional entries at the beginning of the AUTODUCK.FMT file. This way, the local entries will be used in place of the standard ones, and you can avoid specify the /S option each time you run Autoduck.
[/c filename]	Specifies topic log <filename> for the build. The topic log specifies a list of topics that can be linked to and is generally used to determine what type of formatting information to output for a paragraph or text tag (for example, bold if no topic is available, and hypertext link if a topic is a available).
[/x id]	Specifies an extraction expression for the build. Only those topics with **@doc** flags matching the expression are extracted. If no extraction flags are specified, all topics are extracted. For more information on extraction expressions, see “Extraction and Filtering Expressions” and “Conditional Topic and Paragraph Extraction”.
[/d const_name=const_text]	Defines a text constant "const_name" as "const_text". Constants can be referenced in the format file for RTF output. By defining a constant on the command line, you can override a constant defined in the format file. For more information on constants, see the discussion of the **[CONSTANT]** section.

Makefile Entries for Autoduck

The DKOALA example project included with Autoduck uses MAKEDOCS.MAK, a Microsoft Visual C++ makefile. MAKEDOCS.MAK is a generic Autoduck makefile that generates Help and Print documentation files using the set of C/C++ files in the current

project directory.

You can run it on the command line using:

```
NMAKE /f makedocs.mak ProjDir="Project Directory" Project="Project Name"
```

You can also run it as a Visual C++ "custom build" entry, using the following custom build entries:

```
Build Command(s):
 nmake /f makedocs.mak Project="$(WkspName)" ProjDir="$(ProjDir)"

Output Files(s):
 Autoduck\$Project.Hlp
 Autoduck\$Project.Doc
```

MAKEDOCS.MAK

The only entry you need to customize below is the ADTOC entry. Make sure it points to the generic Autoduck CONTENTS.D file or to a custom contents file you have created specifically for your project.

```
# Autoduck MAKEFILE
#
# Eric Artzt, Program Manager
# Consumer Division, Kids Software Group
# Internet  :  erica@microsoft.com
#


OUTDIR  = $(ProjDir)\Autoduck
TARGET  = $(Project)
TITLE   = $(TARGET) Help
DOCHDR  = $(TARGET) API Reference
AD      = autoduck.exe
ADTOC   = "C:\Bin\Contents.D"
ADHLP   = /RH /O$(OUTDIR)\$(TARGET).RTF /D "title=$(TITLE)"
ADDOC   = /RD /O$(OUTDIR)\$(TARGET).DOC /D "doc_header=$(DOCHDR)"
ADTAB   = 8
HC      = hcw /a /e /c
SOURCE  = *.cpp *.h

# Help and Doc targets

target ::
!if !EXIST("$(OUTDIR)")
 md $(OUTDIR)
! endif

target :: $(TARGET).hlp $(TARGET).doc

clean:
 if exist $(OUTDIR)\*.rtf del $(OUTDIR)\*.rtf
 if exist $(OUTDIR)\*.hpj del $(OUTDIR)\*.hpj
 if exist $(OUTDIR)\$(TARGET).doc del $(OUTDIR)\$(TARGET).doc
 if exist $(OUTDIR)\$(TARGET).hlp del $(OUTDIR)\$(TARGET).hlp

# Generate a Help file
```

```
$(TARGET).rtf : $(SOURCE) $(ADTOC)
 $(AD) $(ADHLP) /t$(ADTAB) $(ADTOC) $(SOURCE)

$(TARGET).hlp : $(TARGET).rtf
 $(HC) $(OUTDIR)\$(TARGET).HPJ

# Generate a print documentation file

$(TARGET).doc :  $(SOURCE)
 $(AD) $(ADDOC) /t$(ADTAB) $(SOURCE)
```

Creating Links Across Multiple Help Files

To create hyperlinks across multiple Help files, you need the new Win95/Winnt 3.51 Help compiler (HCW) and viewer (WinHlp32). You will create a Help Contents file that will be referenced by all the help files, and you will edit the HPJ files produced by Autoduck to reference the new .CNT file. Luckily, you only need to do this once, since the HPJ file is not overwritten by Autoduck, and you can save the .CNT file between builds. Make sure your makefile is not deleting the HPJ on a cleanup pass.

Here's how to do it:

Step 1: Construct a Cross-Build Log File

Construct a cross-build log file listing all the topics in all the help files you wish to link. For example:

```
autoduck /n /rh /lproject.log project1\*.cpp project1\*.h
autoduck /n /rh  /a /lproject.log project2\*.cpp project2\*.h
autoduck /n /rh  /a /lproject.log project3\*.cpp project3\*.h
```

Step 2: Create the RTF Files

Build the separate Help RTF files referencing the log file you built earlier. For example:

```
autoduck /rh  /cproject.log project1\*.cpp project1\*.h /oproject1.rtf
autoduck /rh  /cproject.log project2\*.cpp project2\*.h /oproject2.rtf
autoduck /rh  /cproject.log project3\*.cpp project3\*.h /oproject3.rtf
```

Now your Autoduck RTF files are hyperlinked across the various help files. If you had not run the log file build in step (1) and referenced it in step (2), Autoduck would not have generated hyperlinks for the cross-file topic references.

Step 3: Create a Help Contents (.cnt) File

The new Help compiler provides a contents file feature that can be used to build elaborate tables of contents for Help files. In this case, we are only using it to make a list of the Help files we want to associate. You can use the HCW application, or you can edit it yourself.

1. Do File.New to create a new Help Contents file.
2. Click the Index Files button at the bottom of the dialog.
3. In the Index Files dialog, click Add to add a help file to your list. You need to type a short descriptive name and the Help filename. Do this for each help file you want to reference.
4. Save the .CNT file (e.g., in the above example, you might call it "Project.CNT")

The resulting file contains the following lines:

```
:Index Project 1 Help File=project1.hlp
:Index Project 2 Help File=project2.hlp
:Index Project 3 Help File=project3.hlp
```

Step 4: Reference the Contents File in the HPJ File

Use the HCW application to edit the Help project file produced by each Autoduck build. Open the .HPJ file generated by Autoduck and make the following steps:

1 Click the Options button.

2 Click the Files tab.

3 Type the name of the .CNT file you created in step 3.

Step 5: Build the Help Files

Build each help file individually using HCW:

```
hcw /a /c /e project1.hpj
hcw /a /c /e project2.hpj
hcw /a /c /e project3.hpj
```

Generating Topic Indexes

Autoduck 2.0 includes a new **@index** tag that lets you generate topic indexes like the ones displayed on the table of contents pages of Help files. A topic index can appear in any topic.

By default, a topic index lists all topics included in the current build. You can filter the set of topics included by specifying filter expressions, for the topic type, extraction tag set, or both. For more information on extraction expressions, see “Extraction and Filtering Expressions”.

For example, the following **@index** tag displays all **@class** and **@mfunc** topics appearing under the extraction flags PARSE or OUTPUT:

```
//@index Parse and Output | class mfunc | PARSE OUTPUT
```

Default Contents File

Autoduck includes a sample contents file, CONTENTS.D, that you can use as a start. You can place this file in the same directory as AUTODUCK.EXE and include it in your Autoduck builds. You can also customize it to better suit the needs of your projects.

Creating a Module Table of Contents

To create a list of all programming constructs defined in a certain source module, define a unique extraction flag for that module, then include an **@index** tag in the **@module** topic for that module.

The following example shows the **@module** comment from the DKOALA.CPP file included in the Autoduck example project. The extraction flag DKOALA is defined at the top of the file, and the **@index** tag included in the module comment generates an index with title "DKOALA Elements" including all source elements included in the file.

```
// @doc DKOALA
//
// @module DKOALA.CPP - Koala Object DLL Chapter 4 |
//
// Example object implemented in a DLL.  This object supports
```

```
// IUnknown and IPersist interfaces, meaning it doesn't know
// anything more than how to return its class ID, but it
// demonstrates a component object in a DLL.
//
// @head3 DKOALA Elements |
// @index | DKOALA
//
// @normal Copyright (c)1993 Microsoft Corporation, All Rights Reserved
```

Conditional Topic and Paragraph Extraction

You can limit the set of topics extracted in an Autoduck build. You can also code special-case paragraph tags that are only extracted in certain conditions. To define which topics or paragraphs are extracted, you use extraction tokens. Extraction tokens are words identifying a class of topics or paragraphs. For example, you might code some topics as INTERNAL (Microsoft only) and others as EXTERNAL (for release in external documentation).

Associating Extraction Tags with Topics

Use the @DOC tag to associate extraction tokens with topics. The @DOC tag must precede any Autoduck topics in the source file. The @DOC tag names a set of extraction tokens to assign to all following topics.

For example, the following @DOC tag defines EXTERNAL and WAVE tokens for all topics following the tag:

```
// @doc EXTERNAL WAVE
```

The extraction tokens set by an @doc tag remain in effect until the end of the source file, or until the next @doc tag. For example, you can code a single @doc tag at the beginning of the file, and the extraction tokens specified by that tag are used for all Autoduck topics in the source file. To reset the extraction tokens, just add another @DOC tag where you want the new tokens to take effect.

Associating Extraction Tags with Individual Tags

You can also associate extraction tokens with individual topic or paragraph tags. This feature can only be used to exclude topics or paragraphs that would otherwise have been included in a build, given the **@doc** flags set in their area of the source file.

In other words, if you have an entire module labeled as @doc EXTERNAL, and you want to exclude a single topic or paragraph as internal, you can mark that individual tag as INTERNAL and it will be excluded from an EXTERNAL build.

For example, you might define EXTERNAL and INTERNAL tokens to define which topics are for external publication and internal publication. At the paragraph level, you might also define tokens for specific API variations (for example, WIN4J). To associate extraction tokens with a paragraph tag, use the following syntax:

```
//@tagname:(TOKEN [TOKEN...])
```

The following example associates a WIN4J token with an @member paragraph tag:

```
//@flag:(WIN4J) KANJI_ONLY_FLAG | This flag...
//@flag:(WIN4G) GERMAN_ONLY_FLAG | This flag...
```

By specifying WIN4J on the Autoduck command line, you could extract just

KANJI_ONLY_FLAG and omit GERMAN_ONLY_FLAG.

In another example, a topic might contain some paragraphs for internal consumption only:

```
// @doc EXTERNAL

// @func int | QuickFixFunc | This function does something...
//
// @rdesc A pointer to something.
//
// @comm:(INTERNAL) This implementation is flawed and needs to be fixed
// for real next time.
```

Specifying Extraction Tokens on the Command Line

The /X command line option lets you specify an extraction expression defining which topics to extract in the build. For example, the following Autoduck command extracts only those topics that have both the EXTERNAL and WIN4J tags:

```
autoduck /x "EXTERNAL & WIN4J" *.c *.h *.d /okanji.rtf
```

See “Extraction and Filtering Expressions” for details on the expression syntax.

Extraction and Filtering Expressions

Autoduck 2.0 lets you specify topic extraction sets and topic index sets using simple boolean expressions. Expressions can be used in the following places:

- Following the /X command-line option, to specify a subset of topics to extract during a build.
- In the $[index] format-file code, to specify a subset of topic titles to include in a topic index. The $[index] code is exposed to users via the @index tag.

Expression Syntax

Autoduck filtering expressions use a simplified C syntax, using OR and AND operators. You can group subexpressions with parentheses.

OR Operator

You can use three versions of OR operator:

- Pipe (|)
- Comma (,)
- Space (no operator, OR is implied)

The different variants are provided for backwards compatibility with version 1x $[index] codes, and to provide for easy expression entry within Autoduck tag fields, where the pipe symbol works as a field separator.

For example, the following expressions all evaluate to TRUE if any of the tags ONE, TWO, or THREE are matched:

```
ONE TWO THREE
ONE, TWO, THREE
ONE | TWO | THREE
```

AND Operator

The AND operator uses an ampersand (&).

For example, the following expression evaluates TRUE only if all three of the tags ONE, TWO, and THREE are matched:

```
ONE & TWO & THREE
```

Parenthesized Expressions

You can use parentheses to group expressions.

For example, the following expression evaluates TRUE only if the tag EXTERNAL is present, along with any of ONE, TWO, or THREE:

```
EXTERNAL & (ONE TWO THREE)
```

Evaluation Order

Expressions are evaluated from left to right. Parenthesized subexpressions are evaluated first. Neither operator has precedence.

Using Expressions for Topic Extraction

Use the /X command-line option to specify a subset of topics to extract during a build.

For example, the following Autoduck command line extracts only those topics with the tag EXTERNAL and any of the tags ONE, TWO, or THREE:

```
autoduck /x "EXTERNAL & (ONE TWO THREE)"
```

The quotations are required to group the expression as a single command-line option.

Using Expressions for Index Filtering

Use the **@index** tag to specify a topic index. You can filter by tag name and by extraction flag. The @index tag has the following syntax:

// @index <index title> | <topic tag expression> | <extraction flag expression>

You can leave off either of the two expressions, but you must include the field separators.

For example, the following @index tag creates a topic index of @class or @mfunc topics residing under the @doc flag NTSECURITY:

```
// @index NT Security Classes | class mfunc | NTSECURITY
```

Topic Logs

Autoduck is often used to generate RTF for use in online documentation (Help). In help builds, text within paragraph and text tags is often used to generate hypertext links. For example, a function name is marked with an <f function> text tag, in a Help build, Autoduck generates RTF formatting for a hypertext link.

However, a hypertext link needs something to link to, and frequently Autoduck topics reference topics that may not be present in the current build. For example, an Autoduck topic

block might reference a function that is part of a standard system API not included in the current Help project. If a link target is unavailable, you don't want to create a hypertext link, because the link will cause an error in the Help file.

Autoduck provides a topic log feature that lets you test for the existence of a link target, then generate the appropriate formatting depending on whether the link target exists. When you run Autoduck, you can specify a topic log file containing a list of topic names. The formatting file specifies alternate formatting blocks for paragraph or text tags, one used if the link target exists, and the other used if no link target exists.

Autoduck can also generate a topic log file using the topics extracted in the current build. This topic log file can be edited and appended to other logs, to create a multi-build log file. Thus you can build your online documentation files from many different Autoduck builds, each of which references a central log file naming all the available link targets.

The Topic Log File

A topic log is a text file containing a list of topic names, each listed on a separate line terminated by a carriage return/line feed pair.

To reference a topic log, you use the Autoduck /C option, as follows:

autoduck /c mdatopic.log /rh *.cpp *.h

To build a topic log using the list of topics extracted in the current build, you use the Autoduck /L option, as follows:

autoduck /x EXTERNAL /l newfile.log *.cpp *.h

Linking Formatting Specs to the Log File

In the formatting specification file, you can specify alternate formatting information for paragraph and text tags. Autoduck lets you check the log file for a topic name, and use different formatting information depending on whether the topic name exists in the log.

To specify conditions for Autoduck formatting information, you use the tag ".IF" statement, which can be used in formatting blocks for paragraph and text tags and in formatting diagrams.

Defining Tags and RTF Output Strings

Autoduck draws its tag definitions and RTF formatting information from a formatting file. The formatting file defines the autoduck tags used in the source files and specifies RTF text output for those tags. See the provided AUTODUCK.FMT file for examples.

Locating the Formatting File

Since the formatting file defines the complete tagset used within the input files, Autoduck must have access to a formatting file. When searching for a formatting file, Autoduck looks in the location named by the /F command-line flag, if used. Otherwise, Autoduck looks in the current directory; on the search path; and then in the directory in which AUTODUCK.EXE is located.

Adding Supplemental Autoduck Tags

To add new tags to the basic set provided in AUTODUCK.FMT, create your own supplemental formatting file.

A supplemental formatting file is organized the same as AUTODUCK.FMT. You can copy tags or whole sections from AUTODUCK.FMT and modify them as needed. Tags defined in the supplemental formatting file have precedence over those in the main formatting file, so you can "override" the formatting strings or other attributes of standard tags in AUTODUCK.FMT.

Use the /S command option to reference the supplemental formatting file when you run Autoduck. You can use multiple supplemental files.

Sections in the Formatting File

The formatting file is divided into a series of sections. Each section has a heading enclosed in square brackets (for example, **[TOPIC]** or **[PARAGRAPH]**). The sections contain one or more items describing tags or other Autoduck elements.

Sections may be repeated in the formatting file (e.g., you can have multiple **[PARAGRAPH]** sections).

The sections are as follows:

Sections

[FILE]
Defines RTF text to insert at the beginning and end of the output file.

[TOPIC]
Defines topic types. A topic is identified by a unique type name and generates a single block of reference information in the output file. In an Autoduck input file, a topic begins with an @doc tag, followed by a topic tag (defined in the **[TOPIC]** section) identifying the type of information contained in the topic.

[PARAGRAPH]
Defines paragraph types. Paragraphs appear within topics and describe items like function parameters, structure fields, and message flags, comments, examples, and other document elements.

[TEXT]
Defines special text used within paragraphs. Special text items identify interesting phrases or elements (for example, function or structure names) and can have their own formatting attributes (such as bold or hypertext).

[CONSTANT]
Defines string constants referenced by the formatting strings used for file, topic, paragraph, and text elements. Constants are useful for storing RTF formatting text in a central location; for example, you can define a string constant containing RTF formatting codes for an example paragraph and then reference that constant wherever you use an example paragraph. In addition, string constants can be defined or overridden using the /D command line flag, so you can insert build-specific text strings in the RTF output.

[INDEX]
Defines the format of indexes inserted in the output file. An index is a list of topics; in Help, indexes can be used to create hypertext links to a series of related topics.

[DIAGRAM]
Defines diagrams to insert within topics. Diagrams are collections of text drawn from topic paragraphs. You can use diagrams to create syntax diagrams for functions, structures, enumerations, and other language elements.

Comments

You can type comments within the formatting file. Preface any comment lines with a

semicolon typed at the beginning of the line.

Format Strings

Format strings consist of literal output text mixed with special entries that reference fields from the Autoduck tags. In any format-string entry in the formatting file, the format string begins with the first non-blank character following the equal sign of the entry and ends with the first entry, section, or comment found following the formatting entry.

For example, the following entry defines a format string that outputs the text "Field 1:" followed by the contents of field 1 of a tag:

```
.format= Field 1: $1
```

The following special elements can be present in a format string:

$$
Specifies a dollar sign ($) character.

##
Specifies a number (#) character.

$[*name*]
Specifies a diagram *name* defined in the **[DIAGRAM]** section of the formatting file. The diagram is output in place of the $[*name*] code.

$[index*:topic_tag_expr:extr_flag_expr*]
Specifies an index to output. By default, all topic names processed in the build are output in the index. By adding the *:topic_tag_expr* and/or *:extr_flag_expr*, you can specify a subset of topics using a specified tag name or residing under a specified combination of extraction flags. For information on tag or flag expressions, see “Extraction Expressions”.

$(*name*)
Specifies a string constant *name* defined in the **[CONSTANT]** section or passed to Autoduck via the /D command-line argument. The constant string is output in place of the $(*name*) code.

$*n*
Specifies a reference to field number *n* from within the tag. The contents of field number *n* are output in place of the $*n* code. If the source paragraph was identified as an example paragraph in the **[PARAGRAPH]** section, the field text is output in example style.

Field numbers start with 1 and end with the count of fields in the tag.

$*tagname.n*
Specifies a reference to field number *n* from within tag *tagname*. Autoduck searches the topic's tag list for a tag matching *tagname*. If no matching tag is found, nothing is output.

Field numbers start with 1 and end with the count of fields in the topic tag.

#*n*
Specifies a reference to field number *n* from within the tag. The contents of field number *n* are output as a WinHelp/Viewer context string; any non-compliant characters are converted to underscores (_).

If the field contains a substring enclosed in angle brackets (e.g., TemplateFunc<class b>), Autoduck strips the substring (including brackets) from the context string.

$!d
Outputs the current date.

$!f
Outputs the source filename of the tag. Use uppercase $!F to convert the filename to all uppercase; with lowercase $!f, the capitalization scheme of the original filename is used.

If referenced in the **[FILE]** section, this code produces no output.

$!p
Outputs the full path name of the source file from which the tag was extracted. Use $!P to make the path all uppercase.

If referenced in the **[FILE]** section, this code produces no output.

$!l
Outputs the source-file line number of the tag.

$!c
Outputs the topic context string.

$!n
Outputs the topic name.

If referenced in the **[FILE]** section, this code produces no output.

[CONSTANT] Section

This section defines constant strings used elsewhere in the formatting file. Constants are useful for reducing duplication of RTF strings in the formatting file. For example, y

The **[CONSTANT]** section can contain one or more of the following items:

Entries

.OUTPUT=
This item defines the output type for the constants created in later **.DEFINE** statements.

.DEFINE=
Defines a constant string.

Comments

You can use constants to define RTF strings for the standard paragraph styles, then refer to those constants in your tag format strings. The format for a constant reference is as follows:

$(constant_name)

```
For example, the following topic-tag definition references
constants called "reset," "rule", "rh1," and "normal": |

.tag=struct, doc, 2, 50, $1
.order=field comm ex
.pre=$(reset)$(rule)\par
$(reset)$(rh1)$1\par
$[structure]
$(reset)$(normal)$2\par
$(reset)$(normal)Defined in: $!p\par
```

Also, constants can be overridden by values passed on the command line (use the /d option). You can define a constant in the formatting file

See Also

[CONSTANT]

[DIAGRAM] Section

This section defines diagrams. Diagrams are elements built from lists of paragraph tags defined within a topic. For example, a function syntax declaration is a diagram, as is a structure declaration.

In the **[DIAGRAM]** section, you identify the following diagram items:

Entries

."TAG"=
This required item defines the diagram name and specifies the output type.

.PRE=
Specifies a format string to output at the beginning of the diagram.

.POST=
Specifies a format string to output at the end of the diagram.

.FORMATFIRST=
Specifies a format string to use for the first repeating entry in the diagram.

.FORMAT=
Specifies the default format string to use for repeating entries in the diagram. This item is required.

.FORMATLAST=
Specifies a format string to use for the last repeating entry in the diagram.

.CANCELIFPRESENT=
Specifies a list of tag names that, if present in the topic, will prevent the diagram from being generated.

."IF"=
Specifies conditions in which this diagram formatting entry should be used. You can specify multiple "IF" tags; the conditions specified by the multiple .IF tags have an implied OR relationship.

Comments

The bracket ([) preceding the **[DIAGRAM]** section name must appear in the first column (no leading spaces are allowed).

For the .FORMATFIRST, .FORMATLAST, and .FORMAT items, AUTODUCK provides the field information for the paragraph corresponding to the current entry. For the .PRE and .POST items, AUTODUCK provides field information for the topic tag.

Example

The following example creates a function syntax diagram using @param paragraph tags:

```
[diagram]

.tag=function, doc, parm

 Pre-formatting string specifies return value, function name, and
 opening parenthesis
```

```
.pre=\pard \plain $(d_normal){\b $1} {\b $2(}

 Post-formatting string specifies closing parenthesis and paragraph
 mark.

.post={\b )}\par
.formatfirst={\b $1} {\i $2}
.format={\b , $1} {\i $2}
.cancelifpresent=syntax
```

See Also

Format Strings
[DIAGRAM]
CANCELIFPRESENT
TAG
PARAGRAPH-IF

[EXTENSION] Section

This section associates source code language types with filename extensions. The source code language type determines the format of comment blocks within a source file.

In the **[EXTENSION]** section, you identify a series of filename extensions using the following item:

Entries

."EXT"=
Defines a filename extension and associates a language type with the extension.

."GENERICDELIM"=
Defines a generic, single-character comment delimiter to use with a "generic" file type not covered by the standard language types.

Comments

The bracket ([) preceding the EXTENSION section name must appear in the first column (no leading spaces are allowed).

See Also

[EXTENSION]
EXT
GENERICDELIM

[FILE] Section

This section defines the blocks of text that appear at the beginning and end of the output file. This information generally consists of the RTF header, including font table, color table, and style table, and any standard text. You can also include fields for outputting topic indexes.

In the **[FILE]** section, you identify the following file attributes:

Entries

.OUTPUT=

This item defines a new block of file formatting information and specifies the output type for the information.

.PRE=

Specifies a format string to output at the beginning of the output file.

.POST=

Specifies a format string to output at the end of the output file.

Comments

The bracket ([) preceding the FILE section name must appear in the first column (no leading spaces are allowed).

Example

The following **[FILE]** section example defines an RTF header used in a Help topic file:

```
[file]
.output=help
.pre={\rtf1\ansi \deff0\deflang1024

{\fonttbl
.
.   // Font definitions
.
}

{\colortbl;
.
.   // Color definitions
}

{\stylesheet
.
.   // Stylesheet definitions
.
}

\pard\plain $(h_heading_1)
$${\footnote $$ Contents}
+{\footnote + contents:0000}
Contents\par

\pard\plain $(h_indexlink){\uldb Overviews}{\v ctx_overviews}\par
\pard\plain $(h_indexlink){\uldb Modules}{\v ctx_modules}\par
\pard\plain $(h_indexlink){\uldb Classes}{\v ctx_classes}\par
\pard\plain $(h_indexlink){\uldb Functions}{\v ctx_functions}\par
\pard\plain $(h_indexlink){\uldb Messages}{\v ctx_messages}\par
\pard\plain $(h_indexlink){\uldb Types}{\v ctx_types}\par

\page

\pard\plain $(h_heading_1)
#{\footnote # ctx_overviews}
$${\footnote $$ Contents: Overviews}
+{\footnote + contents:0010}
Overviews\par
```

```
$[index:topic]

\page
.
.   // Other header topics
```

See Also

Format Strings
[FILE]
OUTPUT

[INDEX] Section

This section defines the format of WinHelp-style topic indexes. An index consists of a series of topic names. The index can list all topics in the build, or it can list a subset of topics by topic type (for example, @func or @api topics). Indexes are inserted in file formatting strings using the $[INDEX] specifier.

AUTODUCK creates each index entry using the topic name (defined using the **.TAG** item in the **[TOPIC]** section). It also outputs a context string derived from the topic name. In the **[INDEX]** section, you can define the RTF formatting codes surrounding each entry in the index.

The **[INDEX]** section can contain the following entries:

Entries

.OUTPUT=
Defines a new block of index formatting information for Help or print output.

.PRE=
Specifies a format string to output before the index.

.POST=
Specifies a format string to output after the index.

.FORMAT=
Specifies a format string to output for each index entry. Use the $!n and $!c field identifiers to output the topic name and context string, respectively.

.PRENAME=
Obsolete (replaced by **.FORMAT**.

.POSTNAME=
Obsolete (replaced by **.FORMAT**.

.PRECONTEXT=
Obsolete (replaced by **.FORMAT**.

.POSTCONTEXT=
Obsolete (replaced by **.FORMAT**.

Comments

All format entries are optional.

For more information on formatting strings used within indexes as well as the $[INDEX] specifier used to insert an index, see “Format Strings”.

Example

The following example shows an [INDEX] section that defines basic WinHelp links for each index entry. For the printed version, the context strings are hidden:

```
[index]

; Help RTF index
;
.output=help
.format=\pard\plain $(h_indexlink){\uldb $!n}{\v $!c}\par
;
; Doc RTF index
;
.output=doc
.format=\pard\plain $(d_indexlink)$!n
;
; HTML index
;
.output=html
.format=<LI><A HREF=#$!c>$!n</A>
;
```

See Also

Format Strings
[INDEX]
OUTPUT

[PARAGRAPH] Section

This section defines paragraph tags. These tags follow the topic tag and define paragraphs within the topic. Paragraph tags are not associated with a specific type of topic; once defined, they can be used within any type of topic.

In the **[PARAGRAPH]** section, you identify the following paragraph-tag attributes:

Entries

."TAG"=
This required item defines the tag name, the number of fields in the tag, and other characteristics.

."PARSESOURCE"=
This item defines source-parsing capabilities for the tag. Autoduck can retrieve source text declared outside the comment block, provided it is provided in a standard location.

."IF"=
Specifies conditions in which this paragraph tag entry should be used. You can specify multiple "IF" tags; the conditions specified by the multiple .IF tags have an implied OR relationship.

."MAP"=
Maps fields in a paragraph tag to a topic tag defined in the same Autoduck entry.

.PRE=
Specifies a format string to output at the beginning of a series of paragraphs of this type. The **.PRE** item is often used to define a heading for a series of similar paragraphs.

.POST=
Specifies a format string to output at the end of a series of paragraphs of this type.

.FORMAT=
Specifies a format string to output for each paragraph. This item is required.

Comments

The bracket ([) preceding the **[PARAGRAPH]** section name must appear in the first column (no leading spaces are allowed).

See Also

Format Strings
[PARAGRAPH]
PARAGRAPH-IF
MAP
PARSESOURCE
TAG

[TEXT] Section

This section defines text tags for use within the field of a topic or paragraph tag. Special text types can identify interesting elements within text (for example, function or parameter names).

In the **[TEXT]** section, you identify the following items:

Entries

."TAG"=
Defines a new text tag and specifies basic attributes for the tag. This item is required.

.FORMAT=
Specifies a format string to output for the tag. This item is required.

."IF"=
Specifies conditions in which this text tag entry should be used. You can specify multiple "IF" tags; the conditions specified by the multiple .IF tags have an implied OR relationship.

Comments

Text formatting strings can use the field specifiers used for Format Strings. For more information, see "Format Strings".

Example

The following excerpt from a [TEXT] section defines text tags for special symbols, for function tags, and message tags:

```
[text]

; ****************************************
; Symbols
; ****************************************

.tag=cp, both, 0
.format=\'a9
.tag=tm, both, 0
```

```
.format=\'99
.tag=gt, both, 0
.format=>
.tag=lt, both, 0
.format=<
.tag=tab, both, 0
.format=\tab
.tag=nl, both, 0
.format=\line
.tag=cmt, both, 0
.format=//
;
; ****************************************
; Functions
; ****************************************

.tag=f, help, 1
.format={\b $1}
.if=$1=$func.2
.tag=f, help, 1
.format={\uldb $1}{\v #1}
.if=exists($1)
.tag=f, both, 1
.format={\b $1}
;
; ****************************************
; Messages
; ****************************************

.tag=m, help, 1
.format=$1
.if=$1=$msg.1
.tag=m, help, 1
.format={\uldb $1}{\v #1}
.if=exists($1)
.tag=m, both, 1
.format=$1
;
```

See Also

Format Strings
[TEXT]
PARAGRAPH-IF
TAG

[TOKEN] Section

This section defines formatting codes for special characters of a particular output type. Previous versions of Autoduck were hard-coded to output Microsoft Rich Text Format (RTF), prefacing the special RTF control characters \, {, and } with an escape, and outputting \par and \tab for the paragraph and tab symbols, respectively.

Autoduck 2.0 introduces the **[TOKEN]** section, which defines the paragraph, tab, and other control characters for a given type of output.

Entries

."OUTPUT"=

Defines the name for the output type, and specifies a default filename extension for output files of this type.

."CONTEXT"=

Defines which outputting context the tokens apply to. This is used to set different character tokens for different field types (currently, regular fields versus example fields). For example, in HTML output, we output a <PAR> code if we are breaking paragraphs in a regular tag fields, but we don't output a <PAR> code within pre-formatted example fields.

."TOKEN"=

Defines a control character for the output type.

."HIGHCHARMASK"=

Defines a formatting string to use for high-ASCII characters.

Example

This section defines the standard "doc" and "help" output types provided in Autoduck 1.x:

```
[token]
.output=doc,rtf     ; defines "doc" output type, with "rtf" extension
.token=^p,\par      ; paragraph token
.token=^t,\tab      ; tab token
.token=\,\\         ; RTF control characters \, {, and }
.token={,\{
.token=},\}
.highcharmask=\'%x  ; high ascii characters are mapped to \'xx

.output=help,rtf
.token=^p,\par
.token=^t,\tab
.token=\,\\
.token={,\{
.token=},\}
.highcharmask=\'%x
```

This section defines the HTML tokens. Note the use of the **.CONTEXT** entry to cancel use of the <PAR> code to break paragraphs within examples:

```
[token]
.output=html,htm
.token=^p,<P>
.token=^t,&#09;
.token=\,\\
.token=<,&lt;
.token=>,&gt;
.highcharmask=&#%d;
;
; example-specific character tokens
.context=example
.token=^p,
```

See Also

Format Strings
[TOKEN]
HIGHCHARMASK
OUTPUT
CONTEXT

[TOPIC] Section

This section defines topic tags. These tags identify a single documentation unit, or topic. For example, the standard AUTODUCK.FMT file defines topic tags for functions, structures, classes, and other C language elements.

In the **[TOPIC]** section, you identify the following topic-tag attributes:

Entries

."TAG"=
This item defines the tag name, the number of fields in the tag, and other characteristics.

."ORDER"=
This item defines the order in which paragraph tags are output.

."CONTEXT"=
This item defines an alternate identifier (context string) for use in Help.

."PARSESOURCE"=
This item defines source-parsing capabilities for the tag. Autoduck can retrieve source text declared outside the comment block, provided it is provided in a standard location.

.PRE=
Specifies a format string to output at the beginning of the topic.

.POST=
Specifies a format string to output at the end of the topic.

Comments

The bracket ([) preceding the TOPIC section name must appear in the first column (no leading spaces are allowed).

Example

The following **[TOPIC]** section entries define Help and Print versions of an @FUNC tag:

```
[topic]
.tag=func, doc, 3, 20, $2
.order=syntax rdesc parm comm ex
.parsesource=function
.pre=$(reset)$(rule)\par
$(reset)$(heading_1)$2\par
$[function]
$(reset)$(normal)$3\par
$(reset)$(normal)Defined in: $!p\par

.tag=func, help, 3, 20, $2
.order=syntax rdesc parm comm ex
.parsesource=function
.pre=\page
$(reset)$(heading_1)
##{\footnote ## #2}
$${\footnote $$ $2}
K{\footnote K functions; $2}
+{\footnote + functions:0000}
$2\par
```

```
$[function]
$(reset)$(normal)$3\par
$(reset)$(normal)Defined in: $!p\par
```

See Also

Format Strings
[TOPIC]
PARAGRAPH-IF
CONTEXT
ORDER
PARSESOURCE
TAG

[CONSTANT] Section: .DEFINE Entry

This item defines a string constant that can be used in any formatting string used in the formatting file. The item consists of the following fields:

[CONSTANT]
.DEFINE *sName, sText*

Entry Fields

sName
Specifies the name of the constant. Type any name up to 63 characters, with no embedded spaces, tabs, commas, or semicolons.

sText
Specifies the string constant.

Example

The following item defines a string constant named STYLE50 as the text "\s50 \sl240":

```
.define=style50,\s50 \sl240
```

Comments

Constants can also be defined using the /D command-line argument to Autoduck. Constants defined on the command line override constants with the same name defined in the format file.

See Also

[CONSTANT]
DEFINE
OUTPUT

[CONSTANT] Section: .OUTPUT Entry

This item defines the output type for following constant definitions. The **.OUTPUT** item has the following format:

[CONSTANT]
.OUTPUT *sOutputType*

Entry Fields

sOutputType
Specifies the output type for the file formatting block. Use one of the following strings:

doc
Specifies the formatting block is used for paper (Word document) output. The formatting block with this *sOutputType* value is used if the user specifies the /RD command-line flag.

help
Specifies the formatting block is used for help (WinHelp/Viewer topic file) output. The formatting block with this *sOutputType* value is used if the user specifies the /RH command-line flag.

both
Specifies the formatting block is used for both document and help output.

See Also

[CONSTANT]
DEFINE
OUTPUT

[DIAGRAM] Section: .CANCELIFPRESENT Entry

This item defines one or more paragraph tags that can cancel the outputting of the diagram.

[DIAGRAM]
.CANCELIFPRESENT *names*

Entry Fields

names
Specifies one or more paragraph tag names, with multiple names separated by commas.

See Also

Format Strings
[DIAGRAM]
CANCELIFPRESENT
TAG
PARAGRAPH-IF

[DIAGRAM] Section: .TAG Entry

This item defines a new diagram.

[DIAGRAM]
.TAG *name, sOutputType, sParaType*

Entry Fields

name
Specifies the name of the diagram. Type any name up to 63 characters, with no embedded spaces, tabs, commas, or semicolons.

sOutputType
Specifies the output type for the diagram. Use one of the following strings:

sDoc
Specifies paper (Word document) output. The formatting block with this *sOutputType* value is used if the user specifies the /RD command-line flag.

help
Specifies Help (WinHelp/Viewer topic file) output. The formatting block with this *sOutputType* value is used if the user specifies the /RH command-line flag.

both
Specifies both document and help output.

sParaType
This item defines the name of the repeating paragraph tag used within the diagram.

See Also

Format Strings
[DIAGRAM]
CANCELIFPRESENT
TAG
PARAGRAPH-IF

[EXTENSION] Section: .EXT Entry

This item associates a language type with a filename extension.

[EXTENSION]
.EXT *sExtension*, *sLangType*

Entry Fields

sExtension
Filename extension (for example, C, CPP, or BAS). Omit the period.

sLangType
Language type. Use one of the following:

C
C or C++ comment style (// or /*)

ASM
Assembly language comments (;)

BAS
Basic comments (')

Generic
Generic file type. You can use the **.GENERICDELIM** item to define a single-character comment delimiter for generic files not covered by the above language types.

Example

The following **[EXTENSION]** section defines a standard set of filename extensions:

```
[extension]

; Filename extension types
; .ext=<extension_text>, c|asm|bas

.ext=c,c
.ext=cpp,c
.ext=cxx,c
.ext=inl,c
.ext=d,c
.ext=h,c
.ext=hpp,c
.ext=hxx,c
.ext=asm,asm
.ext=bas,bas
.ext=mst,bas
.ext=generic,clw
.genericdelim=!
```

See Also

[EXTENSION]
EXT
GENERICDELIM

[EXTENSION] Section: .GENERICDELIM Entry

This item specifies a single-character comment delimiter to use for generic file types not covered by the standard language types. A single-character comment delimiter is similar to the apostrophe (') comment delimiter in BASIC or the semicolon (;) delimiter in assembly language; anything following the character on that line is considered part of a comment, and can therefore be parsed by Autoduck.

[EXTENSION]
.GENERICDELIM *sDelim*

Entry Fields

sDelim
Comment delimiter character (must be a single character).

Example

The following **[EXTENSION]** section defines a source file type with extension .CLW. Source files with this extension are considered "generic" and have comments delimited with an exclamation point (!).

[extension]

; Filename extension types ; .ext=, c

```
asm|bas

.ext=c,c
... <other extensions>
.ext=clw,generic
.genericdelim=!
```

Therefore, Autoduck would parse the following comment in a .CLW source file:

```
!@doc
!@func return_value | ProcedureName | Description
!@comm general comments about the procedure
!@devnote developer notes about each procedure
```

See Also

[EXTENSION]
EXT
GENERICDELIM

[FILE] Section: .OUTPUT Entry

This item defines a new file formatting block for a specific type of output. Specific formatting information for the output type is specified using **.PRE** and **.POST** items following the **.OUTPUT** item. The **.OUTPUT** item has the following format:

[FILE]
.OUTPUT *sOutputType*

Entry Fields

sOutputType
Specifies the output type for the file formatting block. Use one of the following strings:

doc
Specifies the formatting block is used for paper (Word document) output. The formatting block with this *sOutputType* value is used if the user specifies the /RD command-line flag.

help
Specifies the formatting block is used for help (WinHelp/Viewer topic file) output. The formatting block with this *sOutputType* value is used if the user specifies the /RH command-line flag.

both
Specifies the formatting block is used for both document and help output.

Comments

The period preceding the **.OUTPUT** item text must appear in the first column (no leading spaces are allowed).

See Also

[FILE]
OUTPUT

[INDEX] Section: .OUTPUT Entry

This item defines a new index formatting block and identifies the output type. Specific formatting information for the output type is specified using the formatting-string entries

following the **.OUTPUT** item.

The **.OUTPUT** item has the following field:

[INDEX]
.OUTPUT *sOutputType*

Entry Fields

sOutputType
Specifies the output type for the index formatting block. Use one of the following strings:

doc
Specifies the formatting block is used for paper (Word document) output. The formatting block with this *sOutputType* value is used if the user specifies the /RD command-line flag.

help
Specifies the formatting block is used for help (WinHelp/Viewer topic INDEX) output. The formatting block with this *sOutputType* value is used if the user specifies the /RH command-line flag.

both
Specifies the formatting block is used for both document and help output.

Comments

The period preceding the **.OUTPUT** item text must appear in the first column (no leading spaces are allowed).

See Also

[INDEX]
OUTPUT

.IF Entry

Specifies one or more statements that determine whether a tag should be used in a given situation. By defining multiple formatting blocks, each of which uses different IF statements, you can create variable formatting blocks.

This entry can be used with any tag type, including topic, paragraph, text, and diagrams.

For example, a text tag might reference a function name. You might want to make the name bold, or you might want to make it a hypertext link. If the text tag just references the same function described by the enclosing topic, a link would return the user to the same topic, so the function name should be set to bold instead. If a link is appropriate, you still want to check if there is a destination topic to jump to, so you would check the logging file to see if the named function is listed there.

The **.IF** item can test the following conditions:

- Compare strings in fields: you can check whether a field in a paragraph or text tag matches a field in a topic tag.
- Log file: you can test to see if a topic name is listed in the log file.
- Paragraph tag present: you can test to see if a particular paragraph tag is present within the topic.

- Field empty: you can test to see if a field in the paragraph or text tag is empty.

You can combine various tests in a single “IF” statement. The results of all such tests must evaluate TRUE (implied AND relationship), otherwise AUTODUCK will not use the paragraph-formatting entry, and instead try to use the next formatting block for the paragraph tag.

You can include multiple **.IF** statements within a single tag definition; the statements have an implied OR relationship (if any are true, the tag will be used).

The **.IF** item specifies the following field:

Entry Fields

sConditionals

Specifies one or more conditional statements, separated by commas:

String Comparison:

field-expression1=field-expression2

Compares the value of *field-expression1* to the value of *field-expression2*. Field expressions consist of a mixture of string constants and field references. See the Comments section for details.

For example, you can use a string comparison to check if a function name referenced in a text tag is the same as the function named in the topic tag; if it is, you can code it as bold instead of creating a jump.

Log File Check:

exists(*field-expression*)

Checks to see if the topic names by *field-expression* is listed in the log file.

Log file checks are generally used to verify hypertext links: if a destination topic is named in the log file, create a link; otherwise, don't create a link.

Paragraph Tag Check:

tagexists(*tagname*)

Checks to see if a tag with the specified name was included in the topic. Use this to determine the tag type of the enclosing topic, or to check whether a paragraph tag of the specified type is included in the topic.

This is generally used within function diagrams, to determine whether to output a parameter list or the word "void". It's also used to provide an alternate tag definition to use within certain types of topics.

Field Empty:

fieldempty(*fieldnum*)

Checks to see whether a field number *fieldnum* is empty. Fields are numbered starting at 1.

Used to output an alternative formatting block if a tag field is empty.

Comments

A *field expression* is a combination of tag field references and text literals. In a field expression, you can include any combination of the following constructs (up to six constructs in a field expression):

$n

References field number *n* in the tag.

$<n

References field number *n* in the tag, but strips any template parameter entry (enclosed

in angle brackets) from the end of the field.

$topictag.n
References field number *n* in the topic tag *@topictag*. This construct evaluates to an empty string if the paragraph tag is not contained in a topic block of type *@topictag*.

literal text
Any literal text, except for spaces, tabs, carriage returns, dollar signs ($), semicolons (;), commas (,), closing parentheses ()), and equal signs (=).

Example

Following are two simple examples of **.IF** statements. For more involved examples, see the AUTODUCK.FMT file and refer to the brief comments there for explanations.

The following expression compares the text in the first tag field with the text in the second @FUNC topic-tag field:

```
.if=$1=$func.2
```

The right-hand operand evaluates to a blank if the current topic block is not a @FUNC topic.

The next expression checks to see if a C++ member function is listed in the topic log:

```
exists($1::$2)
```

[PARAGRAPH] Section: .MAP Entry

Maps fields in the paragraph tags to fields in a topic tag defined within the same Autoduck entry.

[PARAGRAPH]
.MAP *sTopicTagname*, *sFieldRef,sFieldRef,...*

Entry Fields

sTopicTagname
Tag name of topic to map.

sFieldRef,sFieldRef,...
One or more field references, each comma-delimited. Each consecutive field reference identifies how to fill in the corresponding field in the topic tag. You can use field references from the current tag or field references from the containing topic tag:

$n
References paragraph tag field "n", where "n" is 1-6.

$t.n
References topic tag field "n", where "n" is 1-6.

Example

The following example maps fields of the @cmember tag to the three topic types that might be defined in tandem:

```
.tag=cmember, help, 4, 2
.pre=$[classhdr]
.format=$(reset)$(term1){\uldb $1}{\v #1} {\uldb $2}{\v #class.1__#2}\par
$(reset)$(def1)$4\par
```

```
.if=exists($class.1::$<2),fieldempty(3),exists($1)
.parsesource=classmember
.map=mfunc,$1,$t.1,$2,$4
.map=enum,$2,$4,$t.1
.map=struct,$2,$4,$t.1
```

See Also

[PARAGRAPH]
PARAGRAPH-IF
MAP
PARSESOURCE
TAG

[PARAGRAPH] Section: .PARSESOURCE Entry

This item defines source-parsing capabilities of the Autoduck tag. Autoduck can parse C source information to obtain fields such as parameter types and names and enumeration types.

When Autoduck determines that fields are missing in a tag, it determines whether source parsing is enabled for the tag. If it is enabled, Autoduck looks at the text immediately preceding the comment block and attempts to parse the missing fields from this text. The missing fields are inserted at the beginning of the tag structure.

There's another ".PARSESOURCE" statement used with topic tags. Essentially, the same statement is used for both topic and paragraph tags; however, the parsing types described in this section apply more closely to paragraph tags.

[PARAGRAPH]
.PARSESOURCE *sParseType*

Entry Fields

sParseType
This field specifies one of the following values indicating the type of source parsing:

parameter
Parameter type and name inserted into fields 1 and 2.

field
Field type and name inserted into fields 1 and 2.

emem
Enumeration name inserted into field 1.

classmember
Member type, name, and (if present) parameter list inserted into fields 1 through 3.

meth
Member type and name inserted into fields 1 and 2.

bparameter
Parameter passing convention (Optional|ByVal|ByRef) inserted into field 1; type and name inserted into fields 2 and 3.

jmethod
Modifiers (optional) - field 1. Type (optional) - field 2. Class name - field 3. Method name - field 4. Parameter list - field 5.

jparameter
Type - field 1. Name - field 2 (including optional brackets).

Example

The following PARSESOURCE item defines parameter parsing:

```
.parsesource=parameter
```

See Also

[PARAGRAPH]
PARAGRAPH-IF
MAP
PARSESOURCE
TAG

[PARAGRAPH] Section: .TAG Entry

This item defines a new paragraph type and includes the following fields:

[PARAGRAPH]
.TAG *sName*, *sOutputType*, *nFields*, *nNestLevel*, *nIsExampleTag*

Entry Fields

sName
Specifies the name of the paragraph. Type any name up to 63 characters, with no embedded spaces, tabs, commas, or semicolons.

sOutputType
Specifies the output type for the paragraph tag. Use one of the following strings:

sDoc
Specifies paper (Word document) output. The formatting block with this *sOutputType* value is used if the user specifies the /RD command-line flag.

help
Specifies Help (WinHelp/Viewer topic file) output. The formatting block with this *sOutputType* value is used if the user specifies the /RH command-line flag.

both
Specifies both document and help output.

nFields
Specifies the number of fields in the tag. Type a number from 1 to 6.

nNestLevel
Specifies the nesting level of the tag. The nesting level defines whether the tag is superior or subordinate to other tags and is used to determine when a series of like tags has started or ended.

Specifically, output text defined by the .PRE item is output when an paragraph tag has the same or lower level (higher *nNestLevel* value) as a preceding tag. Output text defined by the .POST item is output when an paragraph tag has a higher level (lower *nNestLevel* value) than a preceding tag. (In all cases, .PRE or .POST text is only output when a new type of tag is encountered, eg. when going from tag "@foo" to tag "@bar," not when going from tag "@foo" to tag "@foo."

nIsExampleTag
Specifies whether this paragraph contains a code example as its last field. With example paragraphs, AUTODUCK treats field delimiter (|) characters encountered in the last paragraph as literal pipe symbols rather than field delimiters. Also, when inserting the contents of the code-fragment field in the output file, AUTODUCK includes white space (tabs and spaces).

See Also

[PARAGRAPH]
PARAGRAPH-IF
MAP
PARSESOURCE
TAG

[TEXT] Section: .TAG Entry

This item defines a new format tag as well as basic attributes for the format tag. The item includes the following fields:

[TEXT]
.TAG *sName*, *sOutputType*, *nFields*

Entry Fields

sName
Specifies the name of the format tag. Type any name up to 63 characters, with no embedded spaces, tabs, commas, or semicolons.

sOutputType
Specifies the output type for the text tag. Use one of the following strings:

doc
Specifies paper (Word document) output. The formatting block with this *sOutputType* value is used if the user specifies the /RD command-line flag.

help
Specifies help (WinHelp/Viewer topic file) output. The formatting block with this *sOutputType* value is used if the user specifies the /RH command-line flag.

both
Specifies both document and help output.

nFields
Specifies the number of fields in the tag. Type a number from 1 to 6.

See Also

[TEXT]
PARAGRAPH-IF
TAG

[TOKEN] Section: .HIGHCHARMASK Entry

Defines a formatting string for a high-ASCII character.

[TOKEN]
.HIGHCHARMASK *sMask*

Entry Fields

sMask
Formatting mask. Use a C/C++ printf-style formatting string.

See Also

[TOKEN]
HIGHCHARMASK
OUTPUT
CONTEXT
TOKEN

[TOKEN] Section: .OUTPUT Entry

Defines the output type and name.

[TOKEN]
.OUTPUT *sName*, *sExtension*

Entry Fields

sName
Name of the output type, referenced elsewhere in the formatting file and in the /R command-line flag. The types "help" and "doc" are predefined and map to the /Rh and /Rd command line flags.

sExtension
Default filename extension for output type. Used if no output filename is specified on the command line.

[TOKEN] Section: .CONTEXT Entry

Defines a context (field type) for character tokens. All **.TOKEN** and **.HIGHCHARMASK** entries following the **.CONTEXT** tag are used within the specified context.

[TOKEN]
.CONTEXT *sContext*, *sExtension*

Entry Fields

sContext
Field type. Currently there are two valid entries:

body
Regular body text fields. This is the default value and is assumed if no **.CONTEXT** entry has been added.

example
Example text fields.

sExtension
Default filename extension for output type. Used if no output filename is specified on the command line.

Example

The following example uses the **.CONTEXT** entry to define alternate formatting for paragraphs within example text:

```
[token]
.output=html,htm
.token=^p,<P>
.token=^t,&#09;
.token=\,\\
.token=<,&lt;
.token=>,&gt;
.highcharmask=&#%d;
;
; example-specific character tokens
.context=example
.token=^p,
```

[TOKEN] Section: .TOKEN Entry

Defines a control token for the output type, and shows how special characters read from an Autoduck comment block are translated to control sequences in the output file.

[TOKEN]
.TOKEN *chToken*

Entry Fields

chToken
Specifies the token. Type a single character to map to the control sequence specified in the second argument, or type one of the following special control tokens:

^p
Paragraph token: a paragraph token is inserted in place of a a double carriage return found within an Autoduck field, or after every line of an Autoduck example field.

^t
Tab token: inserted in place of a tab. Note that leading tabs are generally stripped from the field.

See Also

[TOKEN]
HIGHCHARMASK
OUTPUT
CONTEXT
TOKEN

[TOPIC] Section: .CONTEXT Entry

This item defines an alternate identifier (context string) for the topic type. Use the CONTEXT item in cases where topics of different tag types might share the same name (for example, you have an object of name FOO and a property of name FOO). The CONTEXT item lets you define a unique identifier for the topic, usually by appending text to the name (for example, FOO_prop).

The context string is used in topic indexes generated for help files and in the topic log generated by Autoduck. If no context string is defined, the topic name as defined in the "TAG" item is used instead.

[TOPIC]
.CONTEXT *sContextNameComponents*

Entry Fields

sContextNameComponents
Specifies the composition of the context string. The context string consists of static text and text drawn from the fields of the topic tag.

The *sContextNameComponents* parameter consists of text intermixed with field references of the format $*n*, where *n* is the field number.

Example

The following CONTEXT item defines a context string for an @PROPERTY tag:

```
.context=$1_prop
```

See Also

[TOPIC]
PARAGRAPH-IF
CONTEXT
ORDER
PARSESOURCE
TAG

[TOPIC] Section: .ORDER Entry

This item defines the order in which paragraph tags are written to the output file.

In the ORDER item, you specify a list of paragraph tag names. These named paragraph types are output in the specified order. Lower-level tags (for example, a paragraph tag with level 2 following a paragraph tag with level 1) are kept together with the higher-level tag. For example, a series of @FLAG tags are output along with the @PARM tags names in the ORDER item.

[TOPIC]
.ORDER

Example

The following ORDER item might be used for a @FUNC tag:

```
.order=rdesc parm ex comm xref
```

This item specifies that the @RDESC tag be output first (including any subordinate tags following @RDESC), followed by @PARM tags, following by @EX tags, and so on.

See Also

[TOPIC]
PARAGRAPH-IF
CONTEXT
ORDER
PARSESOURCE
TAG

[TOPIC] Section: .PARSESOURCE Entry

This item defines source-parsing capabilities of the Autoduck tag. Autoduck can parse C source information to obtain fields such as return type, function name, and class name.

When Autoduck determines that fields are missing in a tag, it determines whether source parsing is enabled for the tag. If it is enabled, Autoduck looks at the text immediately following the comment block and attempts to parse the missing fields from this text. The missing fields are inserted at the beginning of the tag structure.

There's another ".PARSESOURCE" statement used with paragraph tags. Essentially, the same statement is used for both topic and paragraph tags; however, the parsing types described in this section apply more closely to topic tags.

[TOPIC]
.PARSESOURCE *sParseType*

Entry Fields

sParseType
This field specifies one of the following values indicating the type of source parsing:

function
Autoduck parses the return type and function name from the function header immediately following the comment block.

memberfunction
Autoduck parses the return type, class name, and function name from the function header immediately following the comment block.

class
Class name inserted into field 1.

enum
Enumeration name inserted into field 1.

const
Constant type and name inserted into fields 1 and 2.

struct
Structure tag name inserted into field 1 (for C++ style declarations only).

bsub
"Sub" keyword plus any modifiers (Private, Public, etc.) inserted into field 1; subroutine

name inserted into field 2.

bfunc
"Function" keyword plus any modifiers (Private, Public, etc.) inserted into field 1; function name inserted into field 2; function return type (if present) inserted into field 3.

union
Union name inserted into field 1 (for C++ style declarations only).

jclass
Modifiers (optional) - field 1. Name - field 2. Extends (optional) - field 3. Implements (optional) - field 4.

jinterface
Modifiers (optional) - field 1. Name - field 2.

Example

The following PARSESOURCE item defines function parsing:

```
.parsesource=function
```

See Also

[TOPIC]
PARAGRAPH-IF
CONTEXT
ORDER
PARSESOURCE
TAG

[TOPIC] Section: .TAG Entry

This item defines a new topic tag as well as basic attributes for the topic tag. The tag includes the following fields:

[TOPIC]
.TAG *sName*, *sOutputType*, *nFields*, *nSortLevel*, *sTopicNameComponents*

Entry Fields

sName
Specifies the name of the tag. Type any name up to 63 characters, with no embedded spaces, tabs, commas, or semicolons.

sOutputType
Specifies the output type for the topic formatting block. Use one of the following strings:

doc
Specifies the formatting block is used for paper (Word document) output. The formatting block with this *sOutputType* value is used if the user specifies the /RD command-line flag.

help
Specifies the formatting block is used for help (WinHelp/Viewer topic file) output. The formatting block with this *sOutputType* value is used if the user specifies the /RH command-line flag.

both
Specifies the formatting block is used for both document and help output.

nFields
Specifies the number of fields in the tag. Type a number from 1 to 6.

nSortLevel
Specifies the sorting level of the topic. This number (from -32768 to 32767) determines where topics of this type are sorted in relation to other types of topics.

If you specify a negative value, topics of this tag type are sorted in a group, but appear in the same order encountered within the source files.

sTopicNameComponents
Specifies the composition of the topic name. The topic name consists of static text and text drawn from the fields of the topic tag. The topic name is used when sorting topics, to identify topics in error and warning messages, and for constructing unique context strings for topics.

The *sTopicNameComponents* parameter consists of text intermixed with field references of the format $*n*, where *n* is the field number. You can also use $<*n*, which strips a C++ template argument list, if present.

Example

The following item defines a topic tag "foo" for use in Help output. The topic tag has three fields, a sorting weight of 100, and uses the second field as its topic name:

```
.tag=foo, help, 3, 100, $2
```

Given this tag, a valid "foo" topic might be defined as follows:

```
// @doc EXTERNAL
// @foo BAR | MyFoo | This is a foo!
```

The topic name for this block is "MyFoo." The next example defines a topic for documenting C++ member functions:

```
.tag=mfunc, help, 4, 80, $2::$3
```

Given this tag, a valid "mfunc" topic with a topic name of "ClassName::MemberFunction" might be defined as follows:

```
// @doc EXTERNAL
// @foo int | ClassName | MemberFunction | This function...
```

Comments

The period preceding the **.TAG** text must appear in the first column (no leading spaces are allowed).

See Also

[TOPIC]
PARAGRAPH-IF
CONTEXT
ORDER
PARSESOURCE
TAG

@doc

The @doc tag identifies a block of Autoduck source. It must be the first Autoduck tag in a comment block. Any text preceding the @doc tag is ignored.

The @doc tag notifies the Autoduck parser of the presence of Autoduck tag blocks within a source file. The @doc tag also defines Autoduck identifiers used to determine which topics to extract from the source file. The identifiers established by a @doc tag remain in force for all Autoduck topics through the end of the source file or the next @doc tag, whichever comes first.

Syntax

@doc *identifiers*

Comments

The *identifiers* field is a block of text consisting of a whitespace-separated list of keywords to associate with Autoduck topics following the @doc tag. You can use these keywords to determine which topics to extract. The /x command-line option identifies which keywords to process. If the @doc tag names any of the keywords listed in /x command-line option, The topics associated with the @doc tag are extracted.

Example

The following is an example of the @doc tag:

```
//@doc EXTERNAL MIDI_INPUT
```

For more information on @DOC, see “Conditional Topic and Paragraph Extraction”.

Topic Tags

Topic tags identify the beginning of an Autoduck topic block. Topic blocks are delimited by topic tags (beginning a new topic block) or by the end of a documentation block.

The following are the standard Autoduck topic tags:

C Topics

These topic tags are used with C elements:

Tag	Usage
@enum	Enumeration types
@func	functions and macros
@module	Module descriptions
@msg	Messages
@struct	Structures
@type	Typedefs

C++ Topics

These topic tags are used with C++ elements:

Tag	Usage
@class	Classes
@mfunc	Member functions
@mdata	Data members
@mstruct	Structure member
@menum	Enumeration member

@const	Constants

Java Topics

These topic tags are used with Java elements:

Tag	Usage
@jclass	Classes
@jinterface	Interfaces
@jmethod	Methods

OLE2 Topics

These topic tags are used with OLE2 elements:

Tag	Usage
@object	OLE objects - use this to document the primary interface for an object
@interface	OLE interfaces
@method	OLE interface methods
@property	OLE object properties
@event	OLE object events

BASIC Topics

These topic tags are used with Visual Basic elements:

Tag	Usage
@bsub	Visual Basic subroutine
@bfunc	Visual Basic function
@btype	Visual Basic type (structure)

Table of Contents and Overview Topics

These topic tags are used to generate a hierarchical table of contents, and for overviews.

Tag	Usage
@contents1	First-level table of contents page
@contents2	Second-level table of contents page
@topic	Overview topic

Paragraph Tags

Paragraph tags identify elements of a topic such as function parameters, structure fields, comments, examples, and other document elements.

The following are the standard Autoduck paragraph tags:

C Tags

These paragraph tags are used in topic tags describing C constructs (as well as C++ and OLE2 derivatives):

Tag	Usage

@emem	Enumeration members
@field	Structure fields
@flag	Flags (constants)
@parm	Parameters
@parmopt	Parameters with default values
@parmvar	Variable-length parameter list
@rdesc	Return values
@globalv	Global variables (used in @module topic)

C++ Tags

These paragraph tags are used within C++ topics:

Tag	Usage
@access	Access rights (private, protected, public)
@base	Base class name
@cmember	Class members (new auto-parsing tag)
@member	Class members (old member tag)
@syntax	Syntax statements for overloaded member functions
@tcarg	Template class arguments
@tfarg	Template function arguments

Java Tags

These paragraph tags are used within Java topics:

Tag	Usage
@jmeth	Java method declaration
@jparm	Java parameter

OLE2 Tags

These paragraph tags are used within OLE2 topics:

Tag	Usage
@meth	Briefly describes a method within a **@object** topic.
@prop	Briefly describes a property within a **@object** topic.
@eve	Briefly describes an event within a **@object** block
@rvalue	Describes return values
@ilist	Lists names of interfaces supported by a property
@supint	Names an interface within a **@object** block and identify how that object implements the interface.

@supby	Used within a **@method** or **@property** topic to identify a list of objects or interfaces that implement the method or property.
@consumes	Used within a **@object** topic to identify a list of interfaces that the object consumes.

BASIC Paragraphs

These paragraph tags are used with Visual Basic elements:

Tag	Usage
@bparm	Visual Basic parameter
@bfield	Visual Basic type field

Comments and Annotations

These paragraph tags are used to add various types of comments and notes to topics:

Tag	Usage
@comm	Comments
@devnote	Developer notes
@ex	Examples
@group	Subheadings
@todo	Undone work
@xref	Cross references

Miscellaneous

These paragraph tags are used for table of contents and other paragraphs:

Tag	Usage
@index	Creates a topic index.
@subindex	Links to second-level contents pages
@normal	Resets formatting to Normal paragraph style
@head1	Heading level 1
@head2	Heading level 2
@head3	Heading level 3
@end	Ends Autoduck parsing within the comment block

Text Tags

Text tags identify special text strings within a paragraph, such as function names, class names, and special characters.

The following are the text tags:

C Tags

These tags are used for C constructs:

Tag	Usage
<f	Functions
<m	Messages
<t	Structures and enumeration types
<p	Parameters
<e	Structure and enumeration elements

C++ Tags

These tags are used for C++ constructs:

Tag	Usage
<c	Classes
<mf	Member functions
<md	Data members

OLE2 Tags

These tags are used for OLE2 constructs:

Tag	Usage
<o	OLE COM objects
<i	OLE COM interfaces
<om	OLE COM interface methods
<op	OLE COM object properties
<oe	OLE COM object events

Graphics

This tag lets you insert a bitmap file:

Tag	Usage
<bmp	Bitmap graphic file.

Special Characters

These tags represent special characters:

Tag	Usage
<cp	Copyright symbol
<tm	Trademark symbol
<rtm	Registered trademark symbol
<en-	En dash character
<em-	Em dash character
<gt	Greater than symbol
<lt	Less than symbol
<nl	New line character

@access (paragraph-level)

The **@access** tag is used within the **@class** tag to create a subheading that identifies the access rights to a group of items.

Syntax

@access *access_specifier*

Example

The following example uses two **@access** tags as subheadings:

```
//@class This class factory object creates Koala objects.
//
//@base public | IClassFactory

class __far CKoalaClassFactory : public IClassFactory
{
//@access Protected Members

protected:
    //@cmember Reference count.

    ULONG           m_cRef;

//@access Public Members

public:
    //@cmember Constructor.

    CKoalaClassFactory(void);

    //@cmember Destructor.

    ~CKoalaClassFactory(void);
.
.   // More definitions.
.
}
```

See Also

@class

@base (paragraph-level)

The **@base** tag is a paragraph tag used within **@class** comment blocks to specify the base class(es) of a C++ class.

Syntax

@base *access_specifier* | *base_classname*

Comments

You can use as many **@base** tags as necessary.

Example

The following example shows the **@base** tag in use:

```
// @class This class encapsulates a window.
//
// @base public | CCmdTarget

class CWnd : public CCmdTarget
{
public:

 // @cmember This function ...

 HWND GetSafeHwnd() const;
```

See Also

@class

@bfield (paragraph-level)

Documents a Basic type field.

Syntax

@bfield *Name* | *Type* | *Description*

Example

The following examples are equivalent:

```
'@btype | MyType | Example of User-Defined Type
'@bfield i | Integer | An integer.
'@bfield s | String | A string.
'@bfield myString$ | | A string without explicit type name.
'@bfield myInt | | An integer without explicit type name.

Type MyType
 i as Integer
 s as String
 myString$
 myInt
End Type

'@btype Example of User-Defined Type
Type MyType
 i as Integer  '@bfield An integer.
 s as String   '@bfield A string.

 ' @bfield A string without explicit type name.
 myString$

 ' @bfield An integer without explicit type name.
 myInt
```

```
End Type
```

See Also

@btype

@bfunc (topic-level)

The **@bfunc** topic tag documents a Visual Basic function.

Syntax

@bfunc *Modifiers* | *Function Name* | *Return Type* | *description*

Example

The following examples are equivalent:

```
'@bfunc Public | RegGetXLValue | Variant | Get XL value from registry
'@bparm | szSection$ | | Section name
'@bparm | szKey$ | | Key name
'@bparm Optional | vDefaultValues | Variant | Default value if key is missing

Public Function RegGetXLValue(szSection$, szKey$, Optional vDefaultValue As
Variant) As Variant
...
End Function

'@bfunc Get XL value from registry
'@bparm Section name
'@bparm Key name
'@bparm Default value if key is missing

Public Function RegGetXLValue(szSection$, szKey$, Optional vDefaultValue As
Variant) As Variant
...
End Function
```

Comments

Autoduck can extract all the tag fields (except the description) from the subroutine definition in the source file.

See Also

@bparm, **@bsub**

@bparm (paragraph-level)

The **@bparm** paragraph tag documents a Basic subroutine or function parameter.

Syntax

@bparm *Decl_Modifiers* | *Name* | *Type* | *Description*

Example

The following examples are equivalent:

```
'@bfunc Function | RegGetXLValue | Variant | Get XL value from registry
'@bparm | szSection$ | | Section name
'@bparm | szKey$ | | Key name
'@bparm Optional | vDefaultValues | Variant | Default value if key is missing

Function RegGetXLValue(szSection$, szKey$, Optional vDefaultValue As Variant) As
Variant
...
End Function

'@bfunc Get XL value from registry
'@bparm Section name
'@bparm Key name
'@bparm Default value if key is missing

Function RegGetXLValue(szSection$, szKey$, Optional vDefaultValue As Variant) As
Variant
...
End Function
```

Comments

Since Visual Basic does not allow inline comments within function or subroutine declarations, you'll need to place the **@bparm** tags in the body of the function/subroutine header.

See Also

@bfunc, **@bsub**

@bsub (topic-level)

The **@bsub** topic tag documents a Visual Basic subroutine.

Syntax

@bsub *Modifiers* | *Subroutine Name* | *description*

Example

The following examples are equivalent:

```
'@bsub Private | ExcelRegistryExamples | Sets and Retrieves values from the
' Registry

Private Sub ExcelRegistryExamples()
...
End Sub

'@bsub Sets and Retrieves values from the Registry

Private Sub ExcelRegistryExamples()
...
End Sub
```

Comments

Autoduck can extract all the tag fields (except the description) from the subroutine definition in the source file.

See Also

@bfunc, **@bparm**

@btype (topic-level)

Documents a Visual Basic user-defined type, or structure.

Syntax

@btype *Modifiers* | *Type Name* | *description*

Example

The following examples are equivalent:

```
'@btype | MyType | Example of User-Defined Type
'@bfield i | Integer | An integer.
'@bfield s | String | A string.
'@bfield myString$ | | A string without explicit type name.
'@bfield myInt | | An integer without explicit type name.

Type MyType
 i as Integer
 s as String
 myString$
 myInt
End Type

'@btype Example of User-Defined Type
Type MyType
 i as Integer  '@bfield An integer.
 s as String   '@bfield A string.

 ' @bfield A string without explicit type name.
 myString$

 ' @bfield An integer without explicit type name.
 myInt
End Type
```

Comments

Autoduck can extract all the tag fields (except the description) from the subroutine definition in the source file.

See Also

@bfield

@cb (topic-level)

The **@cb** tag is a topic tag used to document C-language callback functions.

Syntax

@cb *type* | *placeholder* | *description*

Paragraph Tags

@rdesc @parm @comm @ex @xref @flag

See Also

@func

@class (topic-level)

The **@class** tag is a topic tag used to document C++ classes.

Syntax

@class *name* | *description*

Example

The following example shows the **@class** tag in use:

```
//@class This class factory object creates Koala objects.
//
//@base public | IClassFactory

class __far CKoalaClassFactory : public IClassFactory
{
//@access Protected Members

protected:
    //@cmember Reference count.

    ULONG           m_cRef;

//@access Public Members

public:
    //@cmember Constructor.

    CKoalaClassFactory(void);

    //@cmember Destructor.

    ~CKoalaClassFactory(void);
.
.   // More definitions.
.
}
```

The following example shows the use of **@class** to document a template class:

```
// @class Template class
//
// @tcarg class | T | A class to store in stack
//
// @tcarg int | i | Initial size of stack

template<class T, int i> class MyStack
{
 // @cmember Top of stack.

 T*  pStack;

 // @cmember Storage of stack items

 T StackBuffer[i];

 // @cmember Count of items in stack

 int cItems = i * sizeof(T);

public:

 // @cmember Constructor for stack.

 MyStack( void );

 // @cmember Adds an item to the stack.

 void push( const T item );

 // @cmember Returns and removes the top item on the stack.

 T& pop( void );
};
```

Comments

Use the **@base** tag to specify base classes. You can use as many **@base** tags as necessary.

To specify a template class, add **@tcarg** paragraph tags to identify the various class template arguments. The presence of **@tcarg** tags cause a template specifier to be printed as the topic title.

See Also

@access @base @cmember @tcarg

@cmember (paragraph-level)

The **@cmember** tag is used within the **@class** tag to provide a simple description of class cmembers. It replaces the earlier **@member** tag.

The tag can parse the first three fields (type, name, and parameter list, if present) from the class member, assuming the tag immediately precedes the line on which the member is defined.

Syntax

@cmember *type* | *name* | *parameter list* | *description*

Comments

The **@cmember** tag is used with in a **@class** topic block to provide brief descriptions of class members.

For class data members, the *parameter list* field is optional.

Use the **@mfunc** and **@mdata** tags to provide complete documentation for member functions and member data. If you define an **@mfunc** or **@mdata** topic matching one of the **@cmember** tags, Autoduck will create a hypertext link (assuming you are pre-building the log file and referencing it using the /C command-line argument.

Example

The following example shows the **@cmember** tag in use:

```
//@class This class factory object creates Koala objects.
//
//@base public | IClassFactory

class __far CKoalaClassFactory : public IClassFactory
{
//@access Protected Members

protected:
    //@cmember Reference count.

    ULONG           m_cRef;

//@access Public Members

public:
    //@cmember Constructor.

    CKoalaClassFactory(void);

    //@cmember Destructor.

    ~CKoalaClassFactory(void);
.
.   // More definitions.
.
}
```

See Also

@class @mfunc @mdata

@comm (paragraph-level)

The **@comm** tag is used to add comments to any Autoduck topic. Unlike other comment tags, the text associated with this tag is included in external (user ed) builds.

Syntax

@comm *comments*

Example

The following example shows the **@comm** tag:

```
// @comm Makes a <c CRect> equal to the intersection of two
// existing rectangles. The intersection is the largest rectangle
// contained in both existing rectangles.
```

See Also

@todo @devnote

@const (topic-level)

The **@const** tag is a topic tag used to document c++ constants.

Syntax

@const *type* | *name* | *description*

Example

The following example shows the **@const** tag:

```
// @const int | iArraySize | Maximum array size.
```

You can also omit the type and name, provided the comment immediately precedes the constant declaration:

```
// @const Maximum array size.

const int iArraySize;
```

@consumes (paragraph-level)

The **@consumes** tag lists OLE interfaces consumed by an object. The tag is used within an **@object** topic block.

Syntax

@consumes *list of interface names*

Example

The following example shows **@consumes** within an **@object** topic block:

```
// @object IgorToolPoolObjServer | This is the MS provided content
// object for the tool pool. It is responsible for maintaining all
// the lists associated with all the tool pool entries - both
// groups and actual elements.
//
// @supint ISpecifyPropertyPages | Property page support
//
// @supint <i IToolPoolEntry> | The means to actually edit the
```

```
// tool pool
//
// @supint <i IToolElemSite> | The means to have all the tool
// pool elements update themselves
//
// @supint IDataObject | Drag/drop & advise support
//
// @supint IDispatch | OLE Automation support
//
// @consumes IMalloc IDispatch <i IEnumTPENTRY>
```

@contents1 (topic-level)

Creates a main contents page. You should only use one **@contents1** topic within your help file. This topic should sort to the top of the RTF file, to be used as the help table of contents.

Syntax

@contents1 | *Contents Heading* | *Contents Paragraph*

Example

The following example, part of the CONTENTS.D file included with Autoduck, creates a first-level contents page for the help file:

```
// @contents1 Contents | To display a list of topics by category, click
// any of the contents entries below. To display an alphabetical list of
// topics, choose the Index button.
```

Comments

Use the **@contents2** and **@subindex** tags to create second-level contents pages and links.

See Also

@subindex @contents2

@contents2 (topic-level)

Creates a second-level contents page. To link to second-level contents pages from the main page, use the **@subindex** paragraph tag. See the CONTENTS.D file, included with Autoduck, for an example.

Syntax

@contents2 | *Contents Heading* | *Contents Paragraph*

See Also

@subindex @contents1

@devnote (paragraph-level)

The **@devnote** tag is used to document developer implementation notes.

Syntax

@devnote *description*

Comments

This tag is for developers and does not generate output for User-Ed Autoduck builds.

See Also

@todo

@emem (paragraph-level)

The **@emem** tag is used to document members of enumeration data types.

Syntax

@emem *name* | *description*

Comments

You can omit the *name* field if the comment block containing the **@emem** tag immediately follows on the same line as the member declaration.

Example

The following example shows how to document enumeration types and members.

```
//@enum Colors.

enum Colors {
 blue,       /@emem The color Blue.
 red,        /@emem The color Red.
};
```

See Also

@enum

@end (paragraph-level)

Empty tag used to terminate the Autoduck section of a comment. Insert the **@end** tag at the end of the Autoduck tags, and any text following the tag will be ignored.

Syntax

@end

@enum (topic-level)

The **@enum** tag is a topic tag used to document enumeration data types.

Syntax

@enum *enumeration_name* | *description*

Example

The following example shows how to document enumeration types and members.

```
//@enum Color values.

enum Colors {
 blue,       //@emem The color Blue.
 red,        //@emem The color Red.
};
```

See Also

@emem

@eve (paragraph-level)

The **@eve** tag names an OLE event supported by an OLE object. The tag is used within an **@object** or **@interface** topic block. Use the *description* field to describe how the object supports the event.

Syntax

@eve *data type* | *event name* | *description*

Example

The following example shows the tag used within a **@object** topic block:

```
//MDA2DCanvasView object

//@object MDA2DCanvasView |The 2D Canvas view object (allows execution of undo-
able

//commands on a 2D Canvas).

//@prop Integer|CameraFitStyle|Determines how a camera view is displayed in a

//canvas view window; one of  CameraOverridesView, StretchToFillView,
ScaleToFillView,

//StretchToViewWidth, StretchToViewHeight, ClipToView

//@prop Boolean|CanCacheView|Determines whether the canvas can be cached to allow

//faster redraw rates.

//@meth HRESULT|CopySelection|Standard clipboard copy.
```

```
//@meth HRESULT|CutSelection|Standard clipboard cut.

//@eve Click|Occurs when the user presses and then releases a mouse button over an
object.

//@eve Deactivate |Occurs before a different canvas view is activated

//@supint IMDA2DCanvasView|For more information, see <o MDA2DCanvasView>.

//@supint IMDA2DGraphicView|Allows access to the view of a generic graphic content
object.
```

@event (topic-level)

The **@event** tag is a topic tag used to document OLE event.

Syntax

@event *interface name*|*event name* | *description*

Example

The following example shows the use of the **@event** tag:

```
//@event IMDA2DCamera|Click|Occurs when the user presses and then releases a mouse
button over
// an object.  It may also occur when the value of a control is changed.
//
//@supby <o MDA2DGroupView>, <o MDA2DPaint>, <o MDA2DMetafile>,
//<o MDA2DRectangle>, <o MDA2DLine>, <o MDA2DLayerView>, <o MDA2DCanvasView>,
//<o MDA2DCamera>
```

@ex (paragraph-level)

The **@ex** tag is used to document example source code. Use the similar **@iex** tag to create an example continuation paragraph.

The second field of the example tag is output as a monospaced paragraph that preserves the spaces and indents from the source file.

Syntax

@ex *description* | *example*

Comments

Text in the *example* field can include special Autoduck characters such as |, <, and > without escaping the characters.

If you use C++ inline comments (//), be sure to place them past the first text column, otherwise the entire line will be omitted from the topic.

Example

The following example uses the **@ex** tag:

```
// @ex The following example adds two objects to a list: |
//
//  CObList list;
//
//  list.AddHead( new CAge( 21 ) );
//  list.AddHead( new CAge( 40 ) ); // List now contains (40, 21);
//  ASSERT( *(CAge*) list.GetTail() == CAge( 21 ) );
```

@field (paragraph-level)

The **@field** tag is used to document structure members.

Syntax

@field *data_type* | *member_name* | *description*

Comments

You can omit the *data_type* and *member_name* fields if the comment block containing the **@field** tag immediately follows on the same line as the member declaration.

Example

The following example shows both usages:

```
// @struct POINT | This structure describes a point.
//
// @field int | x | Specifies the x-coordinate.
//
// @field int | y | Specifies the y-coordinate.

typedef struct tagPOINT
{
 int x;
 int y;
} POINT;

// @struct POINT | This structure describes a point.

typedef struct tagPOINT
{
 int x;    // @field Specifies the x-coordinate.
 int y;    // @field Specifies the y-coordinate.
} POINT;
```

See Also

@flag @struct

@flag (paragraph-level)

The **@flag** tag is used to document constant flags for parameters, return values, and structure fields.

Syntax

@flag *name* | *description*

Example

The following example shows the **@flag** tag (this time used with the **@rdesc** tag):

```
// @func This function compares two strings.
//
// @rdesc Returns one of the following values:
//
// @flag -1 | If <p szStr1> is smaller.
// @flag 1  | If <p szStr2> is smaller.
// @flag 0  | If <p szStr1> and <p szStr2> are the same.

int strcmp(
 char *szStr1,  // @parm Specifies a pointer to the first string.
 char *szStr2)  // @parm Specifies a pointer to the second string.
```

See Also

@parm @field @rdesc

@func (topic-level)

The **@func** tag is a topic tag used to document C-language functions.

Syntax

@func *type* | *name* | *description*

Example

The following shows examples of an @func tag. The first variation shows all the information entered in the tag itself. The second variation lets Autoduck parse information from the function header.

```
// @func int | strcmp | This function compares two strings.
//
// @parm char *| szStr1 | Specifies a pointer to the first string.
//
// @parm char *| szStr2 | Specifies a pointer to the second string.
//
// @rdesc Returns one of the following values:
//
// @flag -1 | If <p szStr1> is smaller.
// @flag 1  | If <p szStr2> is smaller.
// @flag 0  | If <p szStr1> and <p szStr2> are the same.


int strcmp(char *szStr1, char *szStr2)

// @func This function compares two strings.
//
// @rdesc Returns one of the following values:
//
// @flag -1 | If <p szStr1> is smaller.
// @flag 1  | If <p szStr2> is smaller.
```

```
// @flag 0  | If <p szStr1> and <p szStr2> are the same.

int strcmp(
 char *szStr1,  // @parm Specifies a pointer to the first string.
 char *szStr2)  // @parm Specifies a pointer to the second string.
```

Comments

The *type* and *name* fields can both be omitted if the function declaration immediately follows the comment block in which the **@func** tag was used.

Paragraph Tags

@rdesc @parm @comm @ex @xref @flag

See Also

@cb

@globalv (paragraph-level)

The **@globalv** tag is used to document global variables and is generally used inside an **@module** topic.

Syntax

@globalv *type name description*

Example

The following example shows the **@globalv** tag:

```
/* @doc DKOALA
*
* @module DKOALA.CPP - Koala Object DLL Chapter 4 |
*
* Example object implemented in a DLL.  This object supports
* IUnknown and IPersist interfaces, meaning it doesn't know
* anything more than how to return its class ID, but it
* demonstrates a component object in a DLL.
*
* Copyright (c)1993 Microsoft Corporation, All Rights Reserved
*
* @index | DKOALA
*
* @normal Kraig Brockschmidt, Software Design Engineer
* Microsoft Systems Developer Relations
*
* Autoduck example by Eric Artzt (erica@microsoft.com)
*/

//Do this once in the entire build
#define INITGUIDS

#include "dkoala.h"


//@globalv Count number of objects
ULONG       g_cObj=0;
```

```
//@globalv Count number of locks
ULONG       g_cLock=0;
```

@group (paragraph-level)

The **@group** tag is used to add a subheading within any Autoduck topic. You must follow the **@group** paragraph with a tag paragraph to reset the tag type; otherwise, all following paragraphs appear in bold.

Syntax

@group *group heading*

See Also

@todo @devnote

@head1 (paragraph-level)

Inserts a level 1 heading (style "Heading 1").

Syntax

@head1 *Heading Text* | *Paragraph text...*

@head2 (paragraph-level)

Inserts a level 2 heading (style "Heading 2").

Syntax

@head2 *Heading Text* | *Paragraph text...*

@head3 (paragraph-level)

Inserts a level 3 heading (style "Heading 3").

Syntax

@head3 *Heading Text* | *Paragraph text...*

@iex (paragraph-level)

The **@iex** tag creates an example paragraph (a monospaced paragraph that preserves the spaces and indents from the source file).

Syntax

@iex *example*

Comments

Text in the *example* field can include special Autoduck characters such as |, <, and > without escaping the characters.

If you use C++ inline comments (//), be sure to place them past the first text column, otherwise the entire line will be omitted from the topic.

Example

The following example uses the **@iex** tag:

```
// @iex
//  CObList list;
//
//  list.AddHead( new CAge( 21 ) );
//  list.AddHead( new CAge( 40 ) ); // List now contains (40, 21);
//  ASSERT( *(CAge*) list.GetTail() == CAge( 21 ) );
```

@ilist (paragraph-level)

The **@ilist** tag is used to list a series of interfaces supported by a property. Only the names of the supported interfaces appear; not a description of the interface.

Syntax

@ilist *interfaceName, interfaceName, ...*

Example

The following example shows the use of the **@ilist** tag:

```
//@ilist IPixelMap, IPersistStorage, IUnknown
```

See Also

@prop

@index (paragraph-level)

Inserts a topic index. For more information on topic indexes, see “Generating Topic Indexes”.

Syntax

@index *tag-extract-expression* | *topic-extract-expression*

Example

For example, the following **@index** tag displays all **@class** and **@mfunc** topics appearing under the extraction flags PARSE or OUTPUT:

```
//@index class mfunc | PARSE OUTPUT
```

See Also

@contents1 @contents2 @index

@interface (topic-level)

The **@interface** tag is a topic tag used to document OLE interfaces.

Syntax

@interface *name* | *description*

Paragraph Tags

@meth @prop @supby @xref @comm

See Also

@object

@jclass (topic-level)

The **@jclass** tag is a topic tag used to document java classes. All the fields except for the description can be automatically parsed - for better future compatibility, you should encode only the description in the tag.

Syntax

@jclass *modifier_list* | *class_name* | *extends_classname* | *implements_classname_list* | *description*

Example

The following example shows the tag in use:

```
//============================================================================
//@jclass UpdateControls class, whoopee.
//
//============================================================================

public class UpdateControls extends Applet
{
.
.  // field declarations omitted
.

//@jmeth,jmethod UpdateControls class constructor

public updatecontrols()
{
```

```
    .
    . // do stuff
    .
}

//@jmeth,jmethod Information support for the applet

public String getAppletInfo()
{
    .
    . // do stuff
    .
}


//@jmeth,jmethod Paint Handler
//@@jparm Something to paint

public void paint(Graphics g)
{
}

//@jmeth,jmethod
// The loadcontrols() function is called from the VBScript code after the page
// has completely loaded.  VBScript passes in all of the controls so the Java
// class can modify them.

public void loadControls(
    Object htmlList,          //@@jparm the list
    Object htmlButton,        //@@jparm the button
    Object htmlText,          //@@jparm the text
    Object htmlCheckbox)  {   //@@jparm the checkbox
    .
    . // do stuff
    .
}
}
```

See Also

@jclass @jinterface @jmeth @jmethod

@jinterface (topic-level)

The **@jinterface** tag is a topic tag used to document java interfaces. All the fields except for the description can be automatically parsed - for better future compatibility, you should encode only the description in the tag.

Syntax

@jinterface *modifier_list* | *interface_name* | *description*

See Also

@jclass @jinterface @jmeth @jmethod

@jmeth (paragraph-level)

The **@jmeth** tag is a paragraph tag used to document java methods. This tag can be used with the **@jmeth** tag if you want to create a separate topic definition for the method.

All the fields except for the description can be automatically parsed - for better future compatibility, you should encode only the description in the tag.

Syntax

@jmeth [*modifier_list*] | [*type_returned*] | *method_name* | [*throws_name*] | [*parameter_list*] | *description*

Example

The following example shows the tag in use:

```
//============================================================================
//@jclass UpdateControls class, whoopee.
//
//============================================================================

public class UpdateControls extends Applet
{
.
.  // field declarations omitted
.

//@jmeth,jmethod UpdateControls class constructor

public updatecontrols()
{
    .
    . // do stuff
    .
}

//@jmeth,jmethod Information support for the applet

public String getAppletInfo()
{
    .
    . // do stuff
    .
}


//@jmeth,jmethod Paint Handler
//@@jparm Something to paint

public void paint(Graphics g)
{
}

//@jmeth,jmethod
// The loadcontrols() function is called from the VBScript code after the page
// has completely loaded.  VBScript passes in all of the controls so the Java
// class can modify them.
```

```
public void loadControls(
    Object htmlList,          //@@jparm the list
    Object htmlButton,        //@@jparm the button
    Object htmlText,          //@@jparm the text
    Object htmlCheckbox)  {   //@@jparm the checkbox
    .
    . // do stuff
    .
}
}
```

See Also

@jclass @jinterface @jmeth @jmethod

@jmethod (topic-level)

The **@jmethod** tag is a topic tag used to document java methods. This tag is normally used with the **@jmeth** tag - you create a combined tag definition to simultaneously define a paragraph tag within the class (**@jmeth**) and the topic tag to define a separate topic (**@jmethod**).

For auto-parsing, you must use the **@jmeth** paragraph tag. The **@jmethod** topic tag has no built-in parsing capability.

Syntax

@jmethod [*modifier_list*] | [*type_returned*] | *class_name* | *method_name* | [*throws_name*] | *description*

Example

The following example shows the tag in use:

```
//============================================================================
//@jclass UpdateControls class, whoopee.
//
//============================================================================

public class UpdateControls extends Applet
{
.
.  // field declarations omitted
.

//@jmeth,jmethod UpdateControls class constructor

public updatecontrols()
{
    .
    . // do stuff
    .
}

//@jmeth,jmethod Information support for the applet
```

```
public String getAppletInfo()
{
    .
    . // do stuff
    .
}


//@jmeth,jmethod Paint Handler
//@@jparm Something to paint

public void paint(Graphics g)
{
}

//@jmeth,jmethod
// The loadcontrols() function is called from the VBScript code after the page
// has completely loaded.  VBScript passes in all of the controls so the Java
// class can modify them.

public void loadControls(
    Object htmlList,          //@@jparm the list
    Object htmlButton,        //@@jparm the button
    Object htmlText,          //@@jparm the text
    Object htmlCheckbox)  {   //@@jparm the checkbox
    .
    . // do stuff
    .
}
}
```

See Also

@jclass @jinterface @jmeth @jmethod

@jparm (paragraph-level)

The **@jparm** tag is a paragraph tag used to document java method parameters. This tag must be used as a supplemental tag within a **@jmethod** block.

All the fields except for the description can be automatically parsed - for better future compatibility, you should encode only the description in the tag.

Autoduck imposes the following restrictions on comment block placement. See the example for clarification - there are examples of each valid format.

If the parameter declaration immediately follows the method name declaration on the same line, the comment block containing the **@jparm** tag must appear on a line before the method header. See the "paint()" method in the example.

If the parameter declaration is on a line by itself, the **@jparm** tag can appear following the source declaration. See the "loadcontrols()" method in the example.

Syntax

@jparm *type_name* | *parameter_name* | *description*

Example

The following example shows the tag in use:

```
//============================================================================
//@jclass UpdateControls class, whoopee.
//
//============================================================================

public class UpdateControls extends Applet
{
.
.  // field declarations omitted
.

//@jmeth,jmethod UpdateControls class constructor

public updatecontrols()
{
    .
    . // do stuff
    .
}

//@jmeth,jmethod Information support for the applet

public String getAppletInfo()
{
    .
    . // do stuff
    .
}


//@jmeth,jmethod Paint Handler
//@@jparm Something to paint

public void paint(Graphics g)
{
}

//@jmeth,jmethod
// The loadcontrols() function is called from the VBScript code after the page
// has completely loaded.  VBScript passes in all of the controls so the Java
// class can modify them.

public void loadControls(
    Object htmlList,         //@@jparm the list
    Object htmlButton,       //@@jparm the button
    Object htmlText,         //@@jparm the text
    Object htmlCheckbox)  {    //@@jparm the checkbox
    .
    . // do stuff
    .
}
}
```

See Also

@jclass @jinterface @jmeth @jmethod

@mdata (topic-level)

The **@mdata** tag is a topic tag used to document class data members.

Syntax

@mdata *data_type* | *class_name* | *member_name* | *description*

Example

The following example shows the **@mdata** tag in use:

```
//@mdata HWND | CWnd | m_hWnd | Contains the window handle for the
// <c CWnd>.
```

See Also

@class @mfunc @access

@member (paragraph-level)

The **@member** tag is used within the **@class** tag to provide a simple description of class members.

NOTE: The **@cmember** tag is preferred for documenting class members; it can automatically parse the type, name, and parameter list from a class member variable or member function.

Syntax

@member *name* | *description*

Comments

The **@member** tag can only be used within an **@class** topic block. Use the **@mfunc** and **@mdata** tags to provide complete documentation for member functions and member data.

See Also

@cmember

@menum (topic-level)

The **@menum** tag is a topic tag used to document enumeration types defined as members of classes.

Syntax

@menum *class_name* | *enumeration_name* | *description*

Example

The following examples show the **@menum** tag:

```
// @class Example of class with nested constructs.

class CMyClass
```

```
{
public:
//@cmember,mstruct Parsing text structure

struct PARSETEXT
{
char *szBase;     //@@field Base of text to parse
char *szCur;      //@@field Current parsing location
};


//@cmember,menum Parsing types

enum PARSETYPES
{
parseStruct = 1,  //@@emem C structure - gets struct tagname
parseClass,       //@@emem C++ class - gets class name
parseFunc,        //@@emem Function - gets return type and name
};
}
```

Paragraph Tags

@emem

See Also

@cmember @class @emem @mstruct "Nesting Topics Inside Topics"

@meth (paragraph-level)

The **@meth** tag names an OLE method supported by an OLE object. The tag is used within an **@object** topic block. Use the *description* field to describe how the object supports the method.

Syntax

@meth *return value*|*method name* | *description*

@method (topic-level)

The **@method** tag is a topic tag used to document OLE interface methods.

Syntax

@method *return type* | *interface name* | *method name* | *description*

Example

The following example shows the use of the **@method** tag:

```
//@method HRESULT|IMDA2DCanvasView| CopySelection |Standard clipboard copy.
```

Paragraph Tags

@supby @parm @rvalue @ex

@mfunc (topic-level)

The **@mfunc** tag is a topic tag used to document class member functions.

Syntax

@mfunc *return_type* | *class_name* | *function_name* | *description*

Example

The following example shows two variations of the **@mfunc** tag, one using full information typed in the tag fields, and the other using the source parsing feature:

```
//@mfunc void | CString | MakeUpper | This function converts the
// string text to uppercase.

void CString::MakeUpper();

//@mfunc This function converts the string text to uppercase.

void CString::MakeUpper();

//@mfunc Template example with class- and function-level template
// args.
//
//@tfarg class | B | A class to pass
//
//@tcarg class | T | A class to store in stack
//@tcarg int | i | Initial size of stack

template< class T, int i >
MyStack< T, i>::popperlink<class B>( void )
{

}
```

Comments

The *return_type*, *class_name*, and *function_name* fields can all be omitted if the function declaration immediately follows the comment block in which the **@mfunc** tag was used.

See Also

@class @mdata @access @tcarg @tfarg

@module (topic-level)

The **@module** tag is a topic tag used to document source code modules.

Syntax

@module *name* | *description*

Example

The following example shows a module comment:

```
/* @doc DKOALA
```

```
*
* @module DKOALA.CPP - Koala Object DLL Chapter 4 |
*
* Example object implemented in a DLL.  This object supports
* IUnknown and IPersist interfaces, meaning it doesn't know
* anything more than how to return its class ID, but it
* demonstrates a component object in a DLL.
*
* Copyright (c)1993 Microsoft Corporation, All Rights Reserved
*
* @index | DKOALA
*
* @normal Kraig Brockschmidt, Software Design Engineer
* Microsoft Systems Developer Relations
*
* Autoduck example by Eric Artzt (erica@microsoft.com)
*/

//Do this once in the entire build
#define INITGUIDS

#include "dkoala.h"

//@globalv Count number of objects
ULONG       g_cObj=0;

//@globalv Count number of locks
ULONG       g_cLock=0;
```

Comments

This tag is generally just used by developers to record comments for a code module.

See Also

@globalv

@msg (topic-level)

The **@msg** tag is a topic tag used to document Windows-style messages.

Syntax

@msg *name* | *description*

Example

The following example shows the **@msg** tag:

```
// @msg WM_TIMER | This message notifies the window of a timer event.
```

Paragraph Tags

@rdesc @parm @comm @ex @xref @flag

@mstruct (topic-level)

The **@mstruct** tag is a topic tag used to document data structures defined as members of classes.

Syntax

@mstruct *class_name* | *structure_name* | *description*

Example

The following examples show the **@mstruct** tag:

```
// @class Example of class with nested constructs.

class CMyClass
{
public:
 //@cmember,mstruct Parsing text structure

 struct PARSETEXT
 {
 char *szBase;     //@@field Base of text to parse
 char *szCur;      //@@field Current parsing location
 };


 //@cmember,menum Parsing types

 enum PARSETYPES
 {
 parseStruct = 1,  //@@emem C structure - gets struct tagname
 parseClass,       //@@emem C++ class - gets class name
 parseFunc,        //@@emem Function - gets return type and name
 };
}
```

Paragraph Tags

@field @flag

See Also

@cmember @class @field @mstruct “Nesting Topics Inside Topics”

@normal (paragraph-level)

Inserts a body text paragraph (style "Normal").

Syntax

@normal Paragraph text...

@object (topic-level)

The **@object** tag is a topic tag used to document OLE objects.

Syntax

@object *name* | *description*

Example

The following example shows the use of the **@object** tag:

```
// Point2D object
//
// @object   Point2D | Represents a two-dimensional coordinate.
//
// @prop     long | X | X-coordinate (read/write)
//
// @prop     long | Y | Y-coordinate (read/write)
//
// @supint   IPoint2D | Primary interface.
//
// @supint   DPoint2D | Exposes IPoint2D for OLE Automation.
//
// @supint   IDispatch | Equivalent to DPoint2D.
```

Paragraph Tags

@meth @prop @supint @consumes

@parm (paragraph-level)

The **@parm** tag is used to document function and message parameters.

Syntax

@parm *data_type* | *parameter_name* | *description*

Comments

You can omit the *data_type* and *parameter_name* fields if the comment block containing the **@parm** tag immediately follows on the same line as the parameter declaration.

Example

The following example shows both usages:

```
// @func int | strcmp | This function compares two strings.
//
// @parm char *| szStr1 | Specifies a pointer to the first string.
//
// @parm char *| szStr2 | Specifies a pointer to the second string.
//
// @rdesc Returns one of the following values:
//
// @flag -1 | If <p szStr1> is smaller.
// @flag 1  | If <p szStr2> is smaller.
// @flag 0  | If <p szStr1> and <p szStr2> are the same.
```

```
int strcmp(char *szStr1, char *szStr2)

// @func This function compares two strings.
//
// @rdesc Returns one of the following values:
//
// @flag -1 | If <p szStr1> is smaller.
// @flag 1  | If <p szStr2> is smaller.
// @flag 0  | If <p szStr1> and <p szStr2> are the same.

int strcmp(
 char *szStr1,  // @parm Specifies a pointer to the first string.
 char *szStr2)  // @parm Specifies a pointer to the second string.
```

See Also

@parmopt @flag @func @mfunc @method

@parmopt (paragraph-level)

The **@parmopt** tag is used to document optional parameters for functions and member functions.

Syntax

@parm *data_type* | *parameter_name* | *default_value description*

Comments

You can omit all fields except for *description* if the source declaration immediately precedes the **@parmopt** comment block, or if the function declaration follows the comment header.

Example

The following examples shows the various usages:

```
// @mfunc void | MyClass | Foo | My Function Foo
// @parmopt ULONG | a | 1 | [in] value of a
// @parmopt ULONG | b | 2 | [in] value of b

void MyClass::Foo(ULONG a=1,ULONG b=2 )
{ ;}

// @mfunc My Function Foo
void MyClass::Foo(
ULONG a=1,  // @parmopt [in] value of a
ULONG b=2   // @parmopt [in] value of b
)
{ ;}

// @mfunc My Function Foo
// @parmopt [in] value of a
// @parmopt [in] value of b

void MyClass::Foo(ULONG a=1, ULONG b=2 )
{ ;}
```

See Also

@parm @flag @func @mfunc @method

@parmvar (paragraph-level)

The **@parmvar** tag is used to document a variable arguments list.

Syntax

@parmvar *description*

Example

The following example shows how the **@parmvar** tag used within a function block:

```
// @func Prints a bunch of stuff to the console.
//
int strcmp(
 char *szFormat,   // @parm Formatting string with one or more
                   //   variable argument codes.
 ...)              // @parmvar One or more parameters matching
                   //   the argument codes in <p szFormat>.
```

See Also

@parm @func

@prop (paragraph-level)

The **@prop** tag names an OLE property supported by an OLE object. The tag is used within an **@object** topic block. Use the *description* field to describe how the object supports the property.

Syntax

@prop *data type* | *property name* | *description*

Example

The following example shows the tag used within a **@object** topic block:

```
// Point2D object
//
// @object   Point2D | Represents a two-dimensional coordinate.
//
// @prop     long | X | X-coordinate (read/write)
//
// @prop     long | Y | Y-coordinate (read/write)
//
// @supint   IPoint2D | Primary interface.
//
// @supint   IDispatch | Equivalent to DPoint2D.
```

@property (topic-level)

The **@property** tag is a topic tag used to document OLE properties.

Syntax

@property *data type* | *interface name* |*property name* | *description*

Example

The following example shows the use of the **@property** tag:

```
// MDA2DLine property Endpoint1 (r/w)
//
// @propertyPoint2D | IMDA2DLine | Endpoint1 | Coordinate of starting endpoint of
line relative to
// the layer's origin,  in layer coordinate units. (read/write)
//
// @supby      MDA2DLine
//
// @comm    The coordinate value can be in the range -2147483648
//              to 2147483647, inclusive, though some methods that
//              accept Endpoint1 parameters may restrict the value to
//              the range -32768 to 32767.
```

See Also

@supby

@rdesc (paragraph-level)

The **@rdesc** tag is used to document return values of functions and messages.

Syntax

@rdesc *description*

Comments

For functions, the return value type is documented with the **@func** or **@mfunc** tag. For messages, the return value type is implicit—it is the type of the function receiving the message.

Example

The following example shows the **@rdesc** tag:

```
// @func This function compares two strings.
//
// @rdesc Returns one of the following values:
//
// @flag -1 | If <p szStr1> is smaller.
// @flag 1  | If <p szStr2> is smaller.
// @flag 0  | If <p szStr1> and <p szStr2> are the same.

int strcmp(
 char *szStr1,  // @parm Specifies a pointer to the first string.
 char *szStr2)  // @parm Specifies a pointer to the second string.
```

@rvalue (paragraph-level)

The **@rvalue** tag is used to document the HRESULT status codes and their meanings.

Syntax

@rvalue *status code* | *description*

Example

The following example uses the **@rvalue** tag:

```
// @rvalue S_OK | The operation succeeded.
```

@struct (topic-level)

The **@struct** tag is a topic tag used to document data structures.

Syntax

@struct *structure_name* | *description*

Example

The following examples show the **@struct** tag:

```
// @struct POINT | This structure describes a point.
//
// @field int | x | Specifies the x-coordinate.
//
// @field int | y | Specifies the y-coordinate.

typedef struct tagPOINT
{
 int x;
 int y;
} POINT;

// @struct POINT | This structure describes a point.

typedef struct tagPOINT
{
 int x;    // @field Specifies the x-coordinate.
 int y;    // @field Specifies the y-coordinate.
} POINT;
```

Paragraph Tags

@field @flag

See Also

@field @mstruct

@subindex (paragraph-level)

Inserts a link to a second-level index page. Use the **@contents2** tag to create a second-level index page.

Syntax

@subindex | *Subindex Title*

Example

The following tag creates a link to a subindex called "COM Elements":

```
//@subindex COM Elements
```

See Also

@contents1 @contents2 @index

@supby (paragraph-level)

The **@supby** tag lists OLE objects or interfaces that support a method or property. The tag is used within a **@method** or **@property** topic block.

Syntax

@supby *list of objects*

Example

The following example shows **@supby** within a **@property** topic block:

```
//
// @propertyPoint2D | IMDA2DLine | Endpoint1 | Coordinate of starting endpoint of
line relative to
// the layer's origin,  in layer coordinate units. (read/write)
//
// @supby      MDA2DLine
//
// @comm    The coordinate value can be in the range -2147483648
//              to 2147483647, inclusive, though some methods that
//              accept Endpoint1 parameters may restrict the value to
//              the range -32768 to 32767.
```

@supint (paragraph-level)

The **@supint** tag names an OLE interface supported by an OLE object. The tag is used within an **@object** topic block. Use the *description* field to describe how the object supports the interface.

Syntax

@supint *interface name* | *description*

Example

The following example shows the tag used within a **@object** topic block:

```
// Point2D object
//
// @object   Point2D | Represents a two-dimensional coordinate.
//
// @prop     long | X | X-coordinate (read/write)
//
// @prop     long | Y | Y-coordinate (read/write)
//
// @supint   IPoint2D | Primary interface.
//
// @supint   DPoint2D | Exposes IPoint2D for OLE Automation.
//
// @supint   IDispatch | Equivalent to DPoint2D.
```

@syntax (paragraph-level)

The **@syntax** tag is used to document syntax for overloaded C++ member functions.

Syntax

@syntax *syntax_statement*

Comments

If this tag is present in a **@func** or **@mfunc** topic block, the automatically-generated syntax statement is omitted and replaced by the text specified in *syntax_statement*.

Example

The following example shows the **@syntax** tag in use:

```
// @mfunc | CString | CString | Constructs a <mf CString>.
//
// @syntax CString();
// @syntax CString(const CString& stringSrc);
// @syntax CString(char ch, int nRepeat = 1);
// @syntax CString(const char* psz);
// @syntax CString(const char* pch, int nLength);
//
// @parm const CString&| stringSrc | Specifies ...
// @parm char | ch | Specifies..
// @parm int | nRepeat | Specifies...
//
// etc etc.
```

See Also

@mfunc

@tcarg (paragraph-level)

The **@tcarg** tag is used to document template arguments for C++ class templates.

Syntax

@tcarg *data_type* | *argument_name* | *description*

Example

The following example shows the **@tcarg** tag used within class and member function definitions:

```
//@class Template class
//@tcarg class | T | A class to store in stack
//@tcarg int | i | Initial size of stack

template<class T, int i> class MyStack
{ ... }

//@mfunc Template constructor function
//@tcarg class | T | A class to store in stack
//@tcarg int | i | Initial size of stack

template< class T, int i >
MyStack< T, i>::MyStack( void )
{

}
```

See Also

@class @mfunc

@tfarg (paragraph-level)

The **@tfarg** tag is used to document template arguments for C++ member functions and for functions.

Syntax

@tfarg *data_type* | *argument_name* | *description*

Example

The following example shows the **@tfarg** tag used within function and member function definitions:

```
//@func Template function test
//@tfarg class | B | A class.
//@tfarg class | C | Another class.

template<class B, class C>
int TemplateFunc(
 B foo,                 //@parm A Foo
 C bar)                 //@parm A Bar
{
```

```
}

//@mfunc Function template args
//@tfarg class | B | A class to pass
//@tcarg class | T | A class to store in stack
//@tcarg int | i | Initial size of stack

template< class T, int i >
MyStack< T, i>::popperlink<class B>( void )
{

}
```

See Also

@func @mfunc

@todo (paragraph-level)

The **@todo** tag is used to document comments about programming work that is not complete or features that are not implemented.

Syntax

@todo *description*

Comments

This tag is for developers and does not generate output for external (user ed) builds. Use the **@comm** tag to create comments that appear in external builds.

See Also

@devnote

@topic (topic-level)

Creates an overview topic. To create links to a contents topic, use the **<l** text tag.

Syntax

@topic *Topic Heading* | Topic Text

See Also

<l

@type (topic-level)

The **@type** tag is a topic tag used to document data types (generally typedefs).

Syntax

@type *type_name* | *description*

Example

The following example shows the **@type** tag:

```
// @type OLECLIPFORMAT | Standard clipboard format.
```

See Also

@struct

@xref (paragraph-level)

The **@xref** tag is used to document cross references to other related topics.

Syntax

@xref *cross references*

Comments

The *cross references* field is a block of text similar to the *description* field in topic tags. Usually this field consists of a whitespace-separated list of related topics. To properly generate hypertext links in Help, cross references must be properly type formatted.

Example

The following example shows the **@xref** tag:

```
// @xref <c CRect> <c CPoint> <mf CRect.EqualRect>
// <mf CRect.InflateRect>
```

<bmp

The **<bmp** tag lets you insert a bitmap file.

Syntax

<bmp *bitmap filename*>

Example

The following paragraph includes the bitmap C:\DOC\CLASSD.DIB:

```
The following illustration shows the class hierarchy:

<bmp c\:/doc/classd\.dib>
```

Comments

When specifying a full path name, use forward slashes instead of backslashes, and be sure to escape any periods or colons in the path name.

<c

The **<c** tag is used to identify references to classes.

Syntax
<c *class name*>

<cp

The **<cp** tag is used to generate a copyright symbol (©).

Syntax
<cp>

<date

Inserts the date of the Autoduck build.

Syntax
<date>

<e

The **<e** tag is used to identify references to structure members.

Syntax
<e *type name.member name*>

<em-

The **<em-** tag is used to generate an em dash character (—).

Syntax
<em->

See Also
<en-

<en-

The **<en-** tag is used to generate an en dash character (–).

Syntax
<en->

See Also
<em-

<f

The **<f** tag is used to identify references to functions and macros.

Syntax
<f *function name*>

<filename

Inserts the source filename.

Syntax
<filename>

<filepath

Inserts the source file path.

Syntax
<filepath>

<gt

The **<gt** tag is used to generate a greater-than symbol (>).

Syntax
<gt>

See Also
<lt

<i

The **<i** tag is used to identify OLE interface names.

Syntax
<i *interface name*>

<im

The **<im** tag is used to identify references to interface methods.

Syntax
<im *interface method*>

<l

Inserts a hypertext link to an overview topic.

Syntax
<l *overview topic title*>

Comments
Be sure to duplicate the overview topic exactly as it appeared in the **@topic** tag, including embedded spaces and punctuation.

See Also
@topic

<lq

Inserts a left quote.

Syntax
<lq>

See Also
<rq

<lt

The **<lt** tag is used to generate a less-than symbol (<).

Syntax
<lt>

See Also
<gt

<m

The **<m** tag is used to identify references to messages.

Syntax
<m *message name*>

<md

The **<md** tag is used to identify references to class member data.

Syntax
<md *class name*::*member name*>

<mf

The **<mf** tag is used to identify references to class member functions.

Syntax
<mf *class name*::*member function name*>

<nl

The **<nl** tag is used to generate a new line character.

Syntax
<nl>

<o

The **<o** tag is used to identify references to OLE COM objects.

Syntax

<o *object name*>

<oe

The **<oe** tag is used to identify references to OLE2 object events.

Syntax

<om *interface name.event name*>

Comments

You can omit the *interface name* if the event belongs to the same interface as the method, property, or event being described. You must still include the period before the *event name*.

<om

The **<om** tag is used to identify references to methods in OLE COM object interfaces.

Syntax

<om *interface name.method name*>

Comments

You can omit the *interface name* if the method belongs to the same interface as the method, property, or event being described. You must still include the period before the *method name*.

<op

The **<op** tag is used to identify references to properties defined for OLE2 COM objects.

Syntax

<op *interface name.property name* >

Comments

You can omit the *interface name* if the property belongs to the same interface as the method, property, or event being described. You must still include the period before the *property name*.

<p

The **<p** tag is used to identify references to parameters.

Syntax
<p *parameter name*>

<rq

Inserts a right quote.

Syntax
<rq>

See Also
<lq

<rtm

The **<rtm** tag is used to generate a registered trademark symbol (®).

Syntax
<rtm>

See Also
<tm

<t

The **<t** tag is used to identify references to structure and enumeration types.

Syntax
<t *type name*>

<tab

Inserts a tab symbol.

Syntax
<tab>

<tm

The **<tm** tag is used to generate a trademark symbol (™).

Syntax

<tm>

See Also

<rtm