

# Documentation Guide

Autodoc feature

Version 2 Release 6

4<sup>th</sup> May 2009

## Comments & questions

Helmut Scherzer

Giesecke & Devrient  
Prinzregentenstr. 159  
D-81677 Munich

Fon.: +49 89 4119 2084  
Mob: +49 174 313 9891  
Fax: +49 89 4119 1905  
[helmut.scherzer@gi-de.com](mailto:helmut.scherzer@gi-de.com)  
[www.gi-de.com](http://www.gi-de.com)



# Table of contents

<b>1 Development Process</b>	<b>9</b>
1.1 Introduction	9
1.2 General implementation standards	10
1.2.1 Definitions	10
1.2.2 General comments	10
1.3 Code Modularity	10
1.4 'C' specific implementation standards	11
1.4.1 Source File Headers	11
1.4.2 Include strategy	12
1.4.3 Function headers	12
1.4.4 Other tags	14
1.5 Special SmartCard programming techniques	17
1.5.1 Unique return codes	17
1.5.2 ERR_TRY, ERR_THROW, ERR_CATCH	20
1.6 Coding strategy	24
1.6.1 Section strategy	25
1.6.2 Step strategy	26
1.6.3 Indenting comments	27
1.6.4 Function declarations	27
1.6.5 Indent levels and braces	28
1.6.6 Multiple Statements Per Line	29
1.6.7 'goto' statements	30
1.6.8 Code optimization	31
1.6.9 Comments	31
1.6.10 Naming convention	33
1.6.11 Header local defines	35
1.6.12 Errors	36
1.7 ASM specific implementation standards	37
1.7.1 Source File Headers	37
1.7.2 Include strategy	37
1.7.3 External Function Names	37
1.7.4 Function Header	38
1.7.5 Other tags	40
1.7.6 Unique return codes	41
1.7.7 Section strategy	41
1.7.8 Function declarations	43
1.7.9 Function calling conventions	43
1.7.10 Comments	43
1.7.11 Label naming	45
1.7.12 Errors	46
1.7.13 Coding Style	46
1.8 SEQ specific implementation standards	47
1.8.1 Source File Headers	47
1.8.2 Writing test sequences	48
1.9 Final Remark	50
<b>2 AUTODOC Overview</b>	<b>51</b>
2.1 About AUTODOC	51
2.2 AUTODOC Comment Blocks	51
2.3 Features of AUTODOC	51
2.4 AUTODOC Tags	52
2.4.1 AUTODOC Comment Blocks	54
2.4.2 Noise Characters	55
2.4.3 Topics Spanning Multiple Comment Blocks	55
2.5 Source Parsing	56
2.5.1 Supported Source Parsing Configurations	56
2.5.2 Parameter and Structure Field Parsing	56

2.5.3 Function and Member Function Parsing	57
2.5.4 Class Parsing	57
2.5.5 Enumeration Member Parsing	57
2.6 Nesting Topics Inside Topics	57
2.6.1 Tagging Nested Topics	58
2.6.2 Format-File Entries for Nested Topics	58
<b>3 Using AUTODOC.EXE</b>	<b>61</b>
3.1 Makefile Entries for AUTODOC	62
3.2 Generating Topic Indexes	63
3.2.1 Default Contents File	64
3.2.2 Creating a Module Table of Contents	64
3.3 Conditional Topic and Paragraph Extraction	64
3.3.1 Associating Extraction Tags with Topics	64
3.3.2 Specifying Extraction Tokens on the Command Line	65
3.4 Extraction and Filtering Expressions	65
3.4.1 Expression Syntax	66
3.4.2 OR Operator	66
3.4.3 AND Operator	66
3.4.4 Parenthesized Expressions	66
3.4.5 Evaluation Order	66
3.4.6 Using Expressions for Topic Extraction	66
3.4.7 Using Expressions for Index Filtering	66
3.5 Topic Logs	67
3.5.1 The Topic Log File	67
3.5.2 Linking Formatting Specs to the Log File	67
3.6 Defining Tags and RTF Output Strings	68
3.6.1 Locating the Formatting File	68
3.6.2 Adding Supplemental AUTODOC Tags	68
<b>4 Sections in the Formatting File</b>	<b>69</b>
4.1 Comments	69
4.2 Format Strings	70
4.3 [CONSTANT] Section	71
4.4 [DIAGRAM] Section	72
4.5 [EXTENSION] Section	73
4.6 [FILE] Section	73
4.7 [INDEX] Section	75
4.8 [PARAGRAPH] Section	76
4.9 [TEXT] Section	76
4.10 [TOKEN] Section	78
4.11 [TOPIC] Section	79
4.12 [CONSTANT] Section: .DEFINE Entry	80
4.13 [CONSTANT] Section: .OUTPUT Entry	80
4.14 [DIAGRAM] Section: .CANCELIFPRESENT Entry	81
4.15 [DIAGRAM] Section: .TAG Entry	81
4.16 [EXTENSION] Section: .EXT Entry	82
4.17 [FILE] Section: .OUTPUT Entry	82
4.18 [INDEX] Section: .OUTPUT Entry	83
4.19 .IF Entry	83
4.20 [PARAGRAPH] Section: .MAP Entry	85
4.21 [PARAGRAPH] Section: .PARSESOURCE Entry	86
4.22 [PARAGRAPH] Section: .TAG Entry	87
4.23 [TEXT] Section: .TAG Entry	88
4.24 [TOKEN] Section: .HIGHCHARMASK Entry	88
4.25 [TOKEN] Section: .OUTPUT Entry	89
4.26 [TOKEN] Section: .TOKEN Entry	89
4.27 [TOPIC] Section: .CONTEXT Entry	90
4.28 [TOPIC] Section: .ORDER Entry	90
4.29 [TOPIC] Section: .PARSESOURCE Entry	91

4.30 [TOPIC] Section: .TAG Entry .....	92
<b>5 AUTODOC Tags .....</b>	<b>95</b>
5.1 The @doc Tag .....	95
5.2 Topic Tags .....	95
5.2.1 C Topics .....	95
5.2.2 C++ Topics .....	96
5.2.3 OLE2 Topics .....	96
5.2.4 BASIC Topics .....	96
5.2.5 Table of Contents and Overview Topics .....	96
5.3 Paragraph Tags .....	97
5.3.1 C Tags .....	97
5.3.2 C++ Tags .....	97
5.3.3 OLE2 Tags .....	97
5.3.4 BASIC Paragraphs .....	98
5.4 Comments and Annotations .....	98
5.4.1 Miscellaneous .....	98
5.4.2 Additional O/S kernel Tags .....	99
5.4.3 Special Sequence File Tags .....	99
5.5 Text Tags .....	99
5.5.1 C Tags .....	100
5.5.2 C++ Tags .....	100
5.5.3 OLE2 Tags .....	100
5.5.4 Graphics .....	100
5.5.5 Special Characters .....	101
5.6 Tags in alphabetical order .....	101
5.6.1 @access (paragraph-level) .....	101
5.6.2 @base (paragraph-level) .....	102
5.6.3 @bfield (paragraph-level) .....	102
5.6.4 @bfunc (topic-level) .....	103
5.6.5 @bparm (paragraph-level) .....	104
5.6.6 @bsub (topic-level) .....	104
5.6.7 @btype (topic-level) .....	105
5.6.8 @catch (topic-level) .....	106
5.6.9 @class (topic-level) .....	106
5.6.10 @cmember (paragraph-level) .....	108
5.6.11 @comm (paragraph-level) .....	109
5.6.12 @const (topic-level) .....	109
5.6.13 @code (topic level) .....	109
5.6.14 @cont (topic-level) .....	109
5.6.15 @consumes (paragraph-level) .....	110
5.6.16 @contents1 (topic-level) .....	111
5.6.17 @contents2 (topic-level) .....	111
5.6.18 @description (topic-level) .....	111
5.6.19 @devnote (paragraph-level) .....	111
5.6.20 @emem (paragraph-level) .....	112
5.6.21 @end (paragraph-level) .....	112
5.6.22 @enum (topic-level) .....	112
5.6.23 @ex (paragraph-level) .....	113
5.6.24 @excont (topic-level) .....	113
5.6.25 @field (paragraph-level) .....	114
5.6.26 @rc (paragraph-level) .....	115
5.6.27 @func (topic-level) .....	115
5.6.28 @globalv (paragraph-level) .....	116
5.6.29 @group (paragraph-level) .....	116
5.6.30 @head1 (paragraph-level) .....	117
5.6.31 @head2 (paragraph-level) .....	117
5.6.32 @head3 (paragraph-level) .....	117
5.6.33 @iex (paragraph-level) .....	117
5.6.34 @ilist (paragraph-level) .....	117

5.6.35	@index (paragraph-level)	118
5.6.36	@interface (topic-level)	118
5.6.37	@mdata (topic-level)	118
5.6.38	@member (paragraph-level)	118
5.6.39	@menum (topic-level)	119
5.6.40	@method (topic-level)	119
5.6.41	@mfunc (topic-level)	120
5.6.42	@module (topic-level)	120
5.6.43	@msg (topic-level)	121
5.6.44	@mstruct (topic-level)	121
5.6.45	@normal (paragraph-level)	122
5.6.46	@parm (paragraph-level)	122
5.6.47	@pre (paragraph-level)	123
5.6.48	@parmvar (paragraph-level)	124
5.6.49	@result (topic-level)	124
5.6.50	@output (paragraph-level)	124
5.6.51	@rc (paragraph-level)	125
5.6.52	@scall	125
5.6.53	@sequence (paragraph-level)	125
5.6.54	@step (topic-level)	126
5.6.55	@struct (topic-level)	126
5.6.56	@subindex (paragraph-level)	127
5.6.57	@supint (paragraph-level)	127
5.6.58	@syntax (paragraph-level)	127
5.6.59	@tcarg (paragraph-level)	128
5.6.60	@testcase (paragraph-level)	128
5.6.61	@tfarg (paragraph-level)	129
5.6.62	@todo (paragraph-level)	129
5.6.63	@topic (topic-level)	130
5.6.64	@type (topic-level)	130
5.6.65	@xref (paragraph-level)	130
5.7	Text Tags	131
5.7.1	<bmp	131
5.7.2	<c	131
5.7.3	<cp	131
5.7.4	<ct	131
5.7.5	<date	131
5.7.6	<e	131
5.7.7	<ef	132
5.7.8	<em-	132
5.7.9	<en-	132
5.7.10	<ex	132
5.7.11	<ex[1..4]>	132
5.7.12	<f	132
5.7.13	<fc	133
5.7.14	<fig	133
5.7.15	<filename	133
5.7.16	<filepath	133
5.7.17	<ge	133
5.7.18	<gt	134
5.7.19	<iex	134
5.7.20	<im	134
5.7.21	<kw	134
5.7.22	<l	134
5.7.23	<le	134
5.7.24	<lq	135
5.7.25	<lt	135
5.7.26	<m	135
5.7.27	<md	135
5.7.28	<mf	135

5.7.29 <notequal> . . . . .	135
5.7.30 <nex . . . . .	136
5.7.31 <nl . . . . .	136
5.7.32 <or . . . . .	136
5.7.33 <p . . . . .	136
5.7.34 <rc . . . . .	136
5.7.35 <rq . . . . .	136
5.7.36 <rtm . . . . .	137
5.7.37 <sub . . . . .	137
5.7.38 <sup . . . . .	137
5.7.39 <sv . . . . .	137
5.7.40 <t . . . . .	137
5.7.41 <tab . . . . .	137
5.7.42 <tbl . . . . .	137
5.7.43 <tm . . . . .	138
5.7.44 <v . . . . .	138
<b>6 Quick Guide for use with FrameMaker . . . . .</b>	<b>139</b>
6.1 Header Section . . . . .	140
6.2 Code section . . . . .	140
6.2.1 Paragraph tags . . . . .	140
6.2.2 Text tags . . . . .	141
6.3 Special characters and symbols . . . . .	142
6.4 Figures and Tables . . . . .	142
6.4.1 Step 1: Create the figures and tables in the reference file . . . . .	143
6.4.2 Step 2: Export the reference file to MIF . . . . .	143
6.4.3 Step 3: Write the program source(s) . . . . .	143
6.4.4 Step 4: Run AUTODOC with the /b option . . . . .	144
6.5 Equations . . . . .	144
6.6 Updating figures, tables and equations . . . . .	144
6.7 Cross references . . . . .	145
6.7.1 Cross references ex-source . . . . .	145
6.7.2 Cross references in-source . . . . .	145
6.8 System variables . . . . .	146
6.9 Adding system variables . . . . .	146
6.10 Assembler Code Example . . . . .	146
6.11 Assembler Example output . . . . .	149
6.12 Module FASTDIV.AXA . . . . .	149
6.13 _s_long_div . . . . .	150
6.13.1 Step 1: copy b in EEPROM to FAMEX temp memory . . . . .	151
6.13.2 Step 2: save temp vars . . . . .	151
6.14 How to invoke AUTODOC . . . . .	152
6.15 New Faeatures . . . . .	153
<b>7 Quick Guide for use with ID Workbench . . . . .</b>	<b>155</b>
7.1 Header Section . . . . .	155
7.2 Code section . . . . .	155
7.2.1 Paragraph tags . . . . .	155
7.2.2 Text tags . . . . .	156
7.3 Special characters and symbols . . . . .	157
7.4 Figures and Tables . . . . .	157
7.4.1 Step 1: Create the figures and tables in the reference file . . . . .	158
7.4.2 Step 2: Export the reference file to MIF . . . . .	158
7.4.3 Step 3: Write the program source(s) . . . . .	158
7.4.4 Step 4: Run AUTODOC with the /b option . . . . .	159
7.5 Equations . . . . .	159
7.6 Updating figures, tables and equations . . . . .	159
7.7 Cross references . . . . .	160
7.7.1 Cross references ex-source . . . . .	160
7.7.2 Cross references in-source . . . . .	160

7.8 System variables .....	161
7.9 Adding system variables .....	161
7.10 Assembler Code Example .....	161
7.11 Assembler Example output .....	164
7.12 Module FASTDIV.AXA .....	164
7.13 _s_long_div .....	165
7.13.1 Step 1: copy b in EEPROM to FAMEX temp memory .....	166
7.13.2 Step 2: save temp vars .....	166
<b>8 Adobe SDK license agreement.</b> .....	167
<b>9 AUTODOC Design</b> .....	177
9.1 Named Destinations .....	177
9.2 Named Destinations versus Cross References .....	177
9.3 Idee .....	177
9.4 Final Design of AutoXref and AutoMarker .....	178
<b>10 Call Graph Generator</b> .....	181
10.1 Introduction .....	181
10.2 How to generate a call graph .....	182
10.2.1 Command line options .....	183
10.2.2 Function definition .....	185
10.2.3 Layer processing .....	186
10.2.4 Using MACROS.REF .....	186
10.2.5 Using LAYERS.REF .....	187
10.2.6 Generated output files .....	188
10.3 Installation package .....	188
<b>11 Glossary and Abbreviations</b> .....	189
11.1 Abbreviations .....	189
11.2 Glossary .....	190
A.0.1 The Windows character sets .....	196
<b>12 INDEX</b> .....	207



**Author's Note:** The following chapters do not yet comply to some of the documentation rules above (e.g. "Only hyperlinks are blue").

This chapter describes coding standards used throughout the O/S kernel projects. The basic idea of coding standards comes from psychological aspects that cover the fields of

- intuitive perception
- unique understanding
- easy to memorize
- ease of use

Good coding style covers these aspects.

Modern coding standards enhance these standards with the use of **machine-readable, post-processable** features. These features allow other tools to summarize and maintain an entire project. An example of this is the creation of a dependency tree of function calls. Modern coding standards **allow other tools** to provide a more sophisticated **level of analysis** on the source, an argument which becomes particularly important with evaluation and certification of code.

Another rationale for having coding standards comes from many years of experience with both well- and badly-written code. Well-written code is much easier to maintain, debug, and understand.

A third aspect of a common coding-style standard is the goal to establish a kind of '**common language**'. If a function header is always specified with the same structure, programmers may exchange modules with a minimum 'changing of gears' when trying to understand the module of another programmer.

Coding standards establish a level of coding discipline at the cost of personal freedom of a programmer. However, it is exactly that freedom which makes it difficult for a third person to understand two modules written by two different people. Coding standards solve this problem.

Last but not least, there are few topics in the world as sensitive to programmers as the discussion of coding style ("Is indentation better with 2 or 3 blanks?"). Due to that sensitivity, it becomes a holy war trying to achieve an on-the-job consensus within a team of more than two programmers.

The answer to all of these problems is, again, coding standards.

According to the basic law of standards, it is better that there is a 'mediocre standard' than 'individual chaos' - which means, the quality of a standard, and that all people work along with it, is more important than optimal content.

The text below has, however, been created with the intention of having optimum quality in coding standards, in particular, to the psychological aspects of 'perception, understanding, etc.', and is based on years of experience with different flavors of coding style.

To make coding standards successful, it only requires one thing from a team of programmers: USING IT!

---

## 1.2 General implementation standards

---

### 1.2.1 Definitions

**Version**      The version of code for a specific platform/chip/environment. This would change for a major change, for example a change of processor platform.

**Release**      A consolidated, uniform “snapshot” of the source code on a specific date.

---

### 1.2.2 General comments

Wherever possible, source code lines shall be limited to 80 characters in length. This is not relevant for comment lines.

**Note:** The reason comes from the Auto-Documentation feature which should display the code in a readable format. This can only be achieved, if the code lines are not broken when exceeding the right margin.

---

## 1.3 Code Modularity

**Code modularity** means, that a **modules are organized in hierarchical levels** e.g. Application Code, Interface Code, Hardware driver. The rule is, that a higher level may call a lower level, but a lower level should not call a higher level. This will reflect the #include strategy. A hardware driver should for instance not link the .H file of a caller (application).

Code modularity is an important aspect in **certification of a project**. Another valuable aspect of code modularity is, that the ‘lower’ modules are transportable and can be used in other projects since they do not depend on the caller (upper hierarchy).

## 1.4 ‘C’ specific implementation standards

The following sections define the major coding standards for the ‘C’ language.

This paragraph does not describe general coding policies, such as the nesting of include files, prototype definitions, or other common practices. However, some important statements must to be made about aspects of coding style in order to provide a consistent and easy to understand layout.

### 1.4.1 Source File Headers

Every ‘C’ source file shall have a header, as shown here. There is an equivalent header for ‘C’ .h files.

```

/*****
**
**      GuD Confidential
**
**      (C) Copyright Giesecke & Devrient Corporation 2001-2008
**
**      LICENSED MATERIALS - PROPERTY OF GIESECKE & DEVRIENT
**
**      All Rights Reserved
**
**      Origins:  GDM / NB4 Munich
**
*****/
*
*  @modname      (name of the module)
*  @project      (project to which the module belongs)
*  @moddesc      (description of the module)
*
*  @status      Version 1 - Release 2
*
*  @restrictions (any restrictions - if none, say "None known")
*
* Tag Identifier Description
* -----
* @rel Hsc06Mar01 | 0 | Initial Version 1 Release 0
* @crv Asw06Mar02 | Hack, LePuce, ...
* *****/
// @doc
// @decl Declarations | The following includes are made in this module

```

#### @modname

The **@modname** entry describes the **path/filename** of the module. It can be inserted **automatically** by a tool that processes the tags.

#### @project

The **@project** entry describes the project name in order to describe in what ‘larger picture’ the particular module has to be seen. If a module is used in more than one project, then all those projects may be listed optionally. (Example: “**SECCOS**”). It can be inserted **automatically** by a tool processing the tags.

#### @moddesc

A brief description of the module’s function

## @status

The **@status** tag is typically filled by a **post-processing tool**. It reflects the current version/release status of the module. Although this information could be retrieved from version control systems it is considered helpful to reflect this information in the actual module. The entry is typically updated each time a project is released (baseline).

## @restrictions

If there are known re/strictions of the module they should be noted here. I.e. this can be “**not DPA tested**” to indicate that this module does still need some modification. Again this entry is helpful in a post processing process of a project in order to identify open issues.

## @rel

The **@rel** (release) entry is **automatically** generated by an additional tool. It is meant to display a release history in the module. Although this is mainly part of a version control system, this information is considered helpful to trace the history of a source code module. An developer does not need to edit this field since it is automatically generated when a source code package is released.

## @crv

The **@crv** (code review) entry informs about date and participants of a **code review**. The entry documents a successful code review which is also important for certification purposes. The entry is typically created by the owner of a document

## @doc

The **@doc** tag is the start sentinel for the AutoDoc processor. The generation of auto-documentation is not done until a **@doc** tag is encountered.

## @decl

The **@decl** tag is the recommended topic tag in order to start the documentation with a correct headline. Although any other topic tag would be allowed (e.g. **@func**) it is very typical to start with declarations at the start of a module. Therefore the **@decl** tag will be the most popular tag to be used as first topic tag.

---

## 1.4.2

### Include strategy

Include statements shall have one form:

```
#include <filename.H>
```

The “< >” notation is used for those files being placed into the

- **include** (system internal references) and

All other definitions shall be made with the appropriate quotes (“ ”) notation. The development environment shall have **include** in its INCLUDE\_PATH definitions. Therefore, the files included with “< >” do not require a path definition.

Header files shall only be placed into the **include** directories (set in the INCLUDE definition of the compiler) or the component’s own directory. Due to code modularization constraints, modules shall not **include** header files from other modules.

---

## 1.4.3

### Function headers

Function headers shall have a generic layout that allows a structured analysis of the function. The generic structure and some additional tags are required to make the function headers machine readable. This is particular important for *automated code analysis*.

The typical function declaration is described in the following example:

```
Q_RETURN DeleteFile(char* FileName, int FileNameLength);
```

and is coded...

```
↓ Column 1                                                    Column 80 ↓
//=====
// @func Q_RETURN | DeleteFile | deletes a file by its name
//
// @parm char* | FileName | the name of the file characterized as a
//                                zero-delimited ASCII string.
//
// @parm int | FileNameLength | the length of the <v FileName> in bytes
//
// @output The following return codes are generated from this function
//
// @rc RC_OK | Function successfully completed
//
// @rc RC_FileNotExisting_6984 | The file specified by <v FileName>
//                                could not be found
//
// @rc RC_InvalidName_6400 | Invalid characters in filename
//
// @desc The function passes the <v FileName> to the function
//         <xref SearchFile> and expects a valid handle. If found, the
//         handle is passed to <xref fsDeleteFile> and the file is deleted.
//         <Fig StateDiagram>.
//
// @cont <xref StateDiagram> demonstrates the function flow of the module.
//         bla... bla ...
//
//=====
```

The function declaration, as shown here, allows a standardized, machine-readable way of defining functions. In the following, a short rationale is given for each of the parts of the declaration.

**@func** This tag specifies the function type, name, and short description. The short description shall be a line text that would fit into one line of a typical table of contents. A more detailed description shall be given in the **@desc** section later.

#### **A vertical bar shall separate the fields.**

This tag is important to allow evaluation programs to identify a function declaration. The fields shall to be separated by the vertical bar (‘|’) character to allow later interpretation of the function declaration by a post-processing program.

**Note:** **A way to find a function implementation** is to GREP on “| *FuncName* “. Since the **bar** with the name occurs **only within the declaration @func**, the function implementation may be found quickly. Many specialized ASCII editors (e.g., CodeWright) allow effective ‘**grepping**’ through all project files on a GUI level.

**@parm** This tag specifies a function parameter type, name, and description. The description shall explain the use and meaning of the parameter. It is highly recommended to describe the parameters with an “On entry,... On return,...” context in order to clearly identify what is expected when the function is called and/or when the function returns, such as:

```
// @parm *int | Index | On entry, Index contains the current
//         current index into the table; on return, this value
//         is incremented by one.
```

**A vertical bar shall separate the fields.**

There shall be as many **@parm** tags as parameters in the function. **@parm** shall be used only if the function has parameters.

These tags are important for evaluation programs to learn about the function's parameters.

**@output** This tag precedes the explanation of the function's return codes. In very short functions where a description **@desc** tag is missing, a remark can be made here as well to the actual output of the function, however, typically information related to what a function does and modifies should be part of the **@desc** section.

**@rc** These tags list the return codes that are generated by this function. Return codes that are propagated from called subfunctions shall not be listed. This method makes it easier to maintain consistency between the actual code and its return code description.

A vertical bar shall separate the fields.

**@rc** shall be used if the function returns return codes. However, return codes, that the function **propagates** from called subfunctions, **shall not be listed** with **@rc** tags. Hence, whenever a function returns a return code that originates in a called subfunction, a **@rc** tag is not included in the description of the function header, even though it returns a return code.

**Note:** If the entire chain of theoretical return codes of a function is wanted, **automatic tools** can parse this information through the project's files. With help of the **@func** declarations a precise image may be obtained only by parsing the comments (!) of the source.

**@desc** This tag lists the function description. A detailed idea of the function's flow shall be included here.

The tag is important to separate the descriptive part from the machine-readable sections.

The use of **@desc** is recommended, if reasonable.

---

## 1.4.4

### Other tags

Other tags that may be used throughout to indicate special comments.

**<Fig *figname*>**

This text tag links a figure (in this case a state diagram) to the source code documentation. In the actual source there is no point trying to add a real figure, but using the **<Fig *figname*>** construct allows to easily link a figure into the actual documentation output.

**Note:** Technically the actual figures will be held in a particular library document.

**<xref *refname*>**

This text tag allows to establish a cross reference into any other file. Here the cross reference points immediately to the figure above, however any other cross reference can be established. The cross reference is created as hyperlink in the actual documentation, hence a reader may then jump into the actual text the reference points to.

*refname* = DocSpec\_RefLabel

The *refname* is composed by a document identifier *DocSpec* and a reference label *RefLabel*. The *DocSpec* is a keyword which uniquely identifies a document by its *path* and *filename*. (e.g. "Gp24" could be the GlobalPlatform specification version 2.4 ...).

The actual translation is made from a control file which has the advantage that the location of target documents may change and the reference can be updated by tools. Another advantage is, that the *DocSpec* is very short and easy to remember whereas the entire path definition (like other tools suggest) is quite a long writing.

The *RefLabel* identifies a particular part of the target document. There is a generic scheme available.

"Ch4.2" (can also be written as "Ch4p2") indicates a reference to chapter 4.2 of the target document. A valid example of a cross reference would be

`<xref Gp24_Ch4p2>`

will (after processing) directly jump from a PDF file into Chapter 4.2 of the Global Platform specification (also PDF file).

This is easy to remember and does not even require to create complicated hyperlinks through file pathes. The actual resolution is automatically made when the source code is run through the documentation processor.

*Refname* can also be composed of two specially reserved *DocSpec* labels followed by any of the project's function name. This is a very elegant way to cross reference any function in the entire project.

**`<xref IMP_FunctionName>`**

indicates a reference to the implementation documentation which was generated from the target function

**Example:** `<xref IMP_memcpy>`

generates a cross reference into the implementation documentation of the *memcpy( )* function

**`<xref API_FunctionName>`**

indicates a reference to the API documentation which was generated from the target function.

**Example:** `<xref API_memcpy>`

generates a cross reference into the API specification of the *memcpy( )* function.

**`<v varname>`**

The example above also shows some special tags. These tags may optionally be used to switch fonts if auto-documentation tools are used during post-processing. Their application is not mandatory, but may serve special purposes. The specification of these optional tags may be found in other parts of this documentation.

**@dpa** This tag identifies the areas for DPA comments

**@edpa** Since a DPA comment usually pertains to a section of code that may be sensitive to a DPA attack, this tag identifies the end of the sensitive code.

**@vulnerability** This tag identifies the areas for vulnerability comments

**@evulnerability** Since a vulnerability comment usually pertains to a section of code that may be sensitive to vulnerability, this tag identifies the end of the sensitive code.

**@xmp** This tag identifies the area for a code example within a comment

**@exmp** This tag identifies the end of the code example within a comment

Example:

```
//-----  
// @xmp  
//     If MSB of ExpWord == 1 we set  
//  
//         XPTR2 = RPTR2   - input for X**2 is output of multiply  
//         RPTR2 = XPTR2   - output for X**2 is input of multiply  
//  
//     else (MSB of ExpWord == 0), we ignore the multiply  
//  
//         XPTR2 = XPTR2   - input for X**2 is input of multiply  
//         RPTR2 = RPTR2   - output for X**2 is output of multiply  
//  
//     always do ....  
//  
//         XPTR1 = RPTR2  
//         RPTR1 = XPTR2  
//  
//     The above algorithm can be reduced to  
//  
//     if bit == 1  
//         XCH(XPTR2,RPTR2)  
//         XPTR1 = RPTR2  
//         RPTR1 = XPTR2  
// @exmp  
//-----
```

**Note:** **@xmp** and **@exmp** may identically be replaced by **<ex>** **<iex>** tags with exactly the same function.



## 1.5 Special SmartCard programming techniques

The proposed techniques are suggestions. Their application depends on the design of a SmartCard operating system which often cannot be altered due to legacy reasons. Nevertheless some ideas are given here. If followed, however, it is possible to use special tools to improve quality and maintenance.

### 1.5.1 Unique return codes

The assignment of return codes is a general issue in SmartCards and communication interface-related projects.

The O/S kernel projects should internally use unique return codes throughout the entire project. This means, at any place a return code is generated, it shall be different than any other return code generated elsewhere.

Since the unique return code realization is a project and platform- dependent issue, the decision whether and how to realize unique return codes **remains project specific**. The following decisions have currently been made.

#### USHORT return codes

The return code is a USHORT (2 byte) variable. The concept of unique return codes is then realized as follows.

**Bits 15..6** Program Code (10 bits = 1024 values). These values are generated by a post-processing tools (RCG.EXE) that creates these bits automatically to form a unique set of return code. If this tool (RCG.EXE) is included into the MAKE cycle, then a programmer does not need to do the actual assignment. Writing "RC\_<name>\_<code>" into the source will be sufficient to resolve the codes.

**Bits 5..0** SVC Return Code (6 bits = 64 possible codes).

The post processing tool **RCG.EXE** will **automatically generate an include file** like the following:

```
#ifndef _grcH_
#define _grcH_
//-----
// F:\SECCOS\CODE\BASE\FS\FSCREATE.C
//-----
// qFileCreate
#define RC_BadFileSize_6A84 (( 3<<6) | Sw6A84)
#define RC_CannotCreateMF_6A80 (( 118<<6) | Sw6A80)
#define RC_PathTooLong_6981 (( 6<<6) | Sw6981)
#define RC_ReservedFileName_6A80 (( 119<<6) | Sw6A80)
#define RC_CannotCreateAsChildOfEF_6400 (( 99<<6) | Sw6400)
#define RC_FileIdAlreadyUsed_6981 (( 7<<6) | Sw6981)

////////////////////////////////////
// F:\SECCOS\CODE\BASE\FS\FSDELETE.C
//-----
// qFileDeleteByPath
#define RC_CannotDeleteMF_6981 (( 8<<6) | Sw6981)
#define RC_DeleteMustBeEF_6981 (( 9<<6) | Sw6981)

////////////////////////////////////
// F:\SECCOS\CODE\BASE\FS\FSDFNAME.C
//-----
// qFileAssign
#define RC_AssignMustBeDF_6400 (( 100<<6) | Sw6400)
```

```
//-----
// qFileGetDfName
#define RC_DfNameBufferTooShort_6400 (( 101<<6) | Sw6400)

//-----
// fsAssignDf
#define RC_DfNameNotSupported_6400 (( 102<<6) | Sw6400)

//-----
// qFileFindDfName
#define RC_DfNameFullMatchToOtherDf_6981 (( 10<<6) | Sw6981)
#define RC_DfNameCorrectlyAssigned_6000 (( 8<<6) | Sw6000)
#define RC_DfNameAssignedToOtherDf_6000 (( 9<<6) | Sw6000)
#define RC_DfNameAssignedToReturnedMemId_6000 (( 10<<6) | Sw6000)
#define RC_DfHasAnotherName_6000 (( 11<<6) | Sw6000)
#define RC_DfNameNotFound_6A82 (( 11<<6) | Sw6A82)
```

The actual code will only use the defined name (e.g. *RC\_FileNotFound\_6984*).

This allows a later assignment of return codes in one place without giving up the uniqueness of the return codes in the actual code.

The general return code handler in the O/S kernel code may then, depending on a system bit, decide whether to return the entire return code, including the unique part (e.g. (124<<6) | Sw6984), or just the non-unique interface part (e.g., RcSet[Sw6984] = 0x6984).

Another alternative would imply a compile time switch which involves two different .H files for the assignment of verbose names to the return codes.

### 1.5.1.1

#### Naming convention for the RC\_<name> return codes

In general the naming convention should be **RC\_<NameInBiCapitalization>**.

There is an important extension to this rule: Often it happens that the actual ISO return code (e.g. 6982) is known to be the 'official' result of the error. In order to keep that information despite the fact that we are generating unique return codes, this can be reflected in the name.

**RC\_NameInBiCapitalization\_6982**

would be the right name in order to keep the particular expected return code. The **6982** is evaluated in a post-processing tool (**RCG.EXE**) and generates the lower 6 bits such that the return code can be dispatched to return **0x6982**.

### 1.5.1.2

#### Rationale

The **unique return code concept** has already been proven to substantially **decrease debugging time**. It takes a minimum amount of time from the reception of a bad return code until the correct identification of its generator. This is essential for the code-test phase and has been found to save hours of time.

Another aspect of unique return codes is the **test coverage**. Using unique return codes, a tester may verify that he has actually hit a point of interest, even without using an emulator.

Furthermore, it can be **formally verified** that all assigned return codes have been generated at least once in a test. Although this is no proof of comprehensive coverage, this idea may contribute to a **systematic approach of code testing**. Without unique return codes, there will always be an uncertainty from where a return code was generated. Confidence, in the latter case, can only be obtained by setting a breakpoint in the emulator. This, however, is not machine controllable and would not allow verification of all the code in an automated test environment.

Using unique return codes allow the establishment of an **automated test environment** with a specified test coverage. This is important, in particular, with highly-evaluated software.

With the proposed solution, the actual return-code table and the *rc.h* file do not need to be changed using a conditional compile. Both types of return-code information - unique and interface - are contained in the unique return code, and may be, depending on a system bit, interpreted at the interface. The final release will then only use the official interface return codes.

### 1.5.1.3

## Comments

Sections of code that generate error codes shall have comments describing the conditions that will generate the specific errors.

The following example is **unacceptable** coding style for the following reasons:

- There are no braces { } following the **if( )** statement
- The naming convention used is incorrect.
- The return code is not **only** the **defined** name.
- There is more than one return point in the code.
- There are no comments to indicate the reason for the error code.

```
pDesc = PSMobj_Find_Active(UTIL_OS_Ptr_Get_Handle(source)); // b.
if (Desc_1 == 0) // a. b.
    return(PSMERR_READ_3 | QSVChadparm); // c. d. e.
offset = UTIL_OS_Ptr_Arg_Get_Offset(source); // b.
start = offset + length;

if (start < offset || // Wrapped
    start > PSMdesc_Length(pDesc)) // a. b.
    return(RC_LengthOverflow_6F00); // d. e.
```

An **acceptable** style is:

```
ERR_TRY
{
    ///////////////////////////////////
    // @step 1 | Locate source object | <xref psm_ObjFindActive>
    // returns the first active memory handle.
    pDesc = psm_ObjFindActive(UTIL_OSPtrArgGetHandle(Source));

    ///////////////////////////////////
    // @cont If no descriptor was found, the card's resources are
    // exhausted and the function aborts
    if(pDesc == 0)
    {
        ERR_THROW(RC_ErrRead3_6400);
    }
    Offset = UTIL_OsPtrArgGetOffset(Source);
    Start = Offset + Length;

    ///////////////////////////////////
    // @step 2 | Check plausibility | Return error if block to read is
    // not completely within object
    if(Start < Offset ||
        Start > psm_DescLength(pDesc))
    {
        ERR_THROW(RC_ErrRead4_6400);
    }
}
```

```

}
ERR_CATCH
{
    return rc;
}

```

## 1.5.2 ERR\_TRY, ERR\_THROW, ERR\_CATCH

The use of ERR\_TRY-ERR\_THROW-ERR\_CATCH was designed to be similar to the syntax used in the C++ environment. During the discussion of coding standards, it turned out that less 'hidden variables' and a slightly higher level of comfort can be realized if the C++ syntax is not the highest priority.

For the purpose of code optimization, three macros are provided. These macros are defined as follows:

```

#define ERR_TRY

#define ERR_EXCEPTION    USHORT rc

#define ERR_THROW(a)    { \
                        rc = a;    \
                        goto ExitLabel; \
                        }

#define ERR_CATCH        { \
                        rc = QSVCgood; \
                        ExitLabel: \
                        }

```

The **ERR\_EXIT** macro was designed for situations where it is not desired to automatically generate the **rc = RC\_OK** statement. It can be used as an alternative to **ERR\_CATCH**.

```

#define ERR_EXIT        { \
                        ExitLabel: \
                        }

```

The following example demonstrates the use of these macros for the code shown above.

```

#include <except.h>

void MyFunc(void)
{
    ERR_EXCEPTION;
    int idx;
    int NewVar;

    ERR_TRY
    {
        NewVar = 0;    // set newvar to zero to allow unconditional 'free'
        for(idx = 0; idx < N; idx++)
        {
            rc = DoSomething(idx);
            if(rc)    // easy propagation of error situations
                ERR_THROW(rc);
        }
        if(FileExists(fn))
            ERR_THROW(rcFS_FileAlreadyExists);
    }
}

```

```

:
}
ERR_CATCH
{
    free(NewVar);
    return rc;    // RetCode is set when a THROW(x) is executed
}
}

```

The example above demonstrates the use of the **ERR\_THROW** and **ERR\_CATCH** macros.

#### **ERR\_TRY**

current defined as empty macro. However in future design it may serve as a special construct to counterfy side channel attacks. It can then serve as a jump address to prevent a function to be tampered by flash / light or other attacks.

The use of **ERR\_TRY** is consistent with the use of try in C++ and therefore quite handy and useful to programmers.

#### **ERR\_EXCEPTION**

needs to be declared only once within a function, without parameters, prior to the use of **ERR\_THROW** and **ERR\_CATCH**. It defines the necessary variable *rc* to be used with the exception handling.

#### **ERR\_THROW( USHORT ReturnCode )**

defines the exit to the (only) exception handler. An **ERR\_THROW( )** statement jumps immediately to the **ERR\_CATCH** section. The parameter *ReturnCode* is assigned to the internally-defined local variable *rc*. *rc* can be used later to propagate the return code that was “thrown” while the actual return-code *ReturnCode* may be use for other purposes in the **ERR\_CATCH** section.

**ERR\_CATCH { }** The **ERR\_CATCH { }** section contains the all code that is the target of the **ERR\_THROW( )** statements.

**ERR\_EXIT { }** The **ERR\_EXIT { }** section contains the all code that is the target of the **ERR\_THROW( )** statements.

The use of these macros is derived from their corresponding use in ‘C++’ code. However, they are ‘C’ code and hence some restrictions apply:

- An **ERR\_CATCH** section must always be present in the same function body.
- Only one **ERR\_CATCH** section is allowed in a function.
- Only one parameter of type **USHORT** may be used with the **ERR\_THROW** statement.
- Code falling into the E\_CATCH section will return with *RetCode* = **RC\_OK**.
- **ERR\_CATCH** shall be the last of the three statements in the function. The normal code-flow may fall into **ERR\_CATCH** or do its own return statement before **ERR\_CATCH**. There is no way of programming to reach any code behind the **ERR\_CATCH** label (except using the forbidden ‘goto’). If another choice is preferred, the **TRY-THROW-CATCH** method should be used instead.
- When memory is allocated, a general cleanup function can be included in the **ERR\_CATCH** section. This guarantees proper housekeeping of memory when an error occurs during code execution. See also 1.4 “‘C’ specific implementation standards”.

## Rationale

Having worked with **ERR+TRY-ERR\_THROW-ERR\_CATCH**, we found that we can do a slight improvement for many typical cases. Falling into **ERR\_CATCH** allows the deletion of allocated objects in the same part of code as an error handler. Falling into **ERR\_CATCH** will automatically generate a **rc = RC\_OK** assignment. Therefore falling into **ERR\_CATCH** can be considered as 'good' exit.

In many cases, it is usual to allocate memory in the module and to free the allocated space either in the case of error or at the end of the function.

Using the **ERR\_CATCH** fall-into mechanism allows effective use of this function.

### 1.5.2.1

#### **ERR\_TRY, ERR\_THROW, ERR\_CATCH Example**

An example of a typical resource allocation is the use of *malloc( )*. According to 'C++' conventions, the corresponding release function (e.g., *free( )*) should immediately return *if it finds an input parameter equal zero*. The following example provides the appropriate idea.

```
void ExampleFunction(void)
{
    ERR_EXCEPTION;

    char *ptr;

    ERR_TRY
    {
        ptr = 0;    // ptr is set to zero to allow unconditional use of free()
        :
        :
        if(SomeError)
            ERR_THROW(rcXX_SomethingIsWrong)           // ptr is still zero

        if(...)
        {
            ptr = malloc(100);
            :
            :
            if(SomeError)
                ERR_THROW(rcXX_SomethingStrangeHappened);
            :
            :
            free(ptr);    // ptr is freed correctly
            ptr = 0;      // and set to zero to avoid trouble with next exception
            :             // another malloc() may follow..., other THROWS too
            :
        }
    }
    ERR_CATCH
    {
        free(ptr);      // as free accepts ptr=0 this can be made
        return RetCode;
    }
    return 0;
}
```

The variable *ptr* is set to zero right at the beginning. This prevents the *free( )* function in the **ERR\_CATCH { }** section from freeing an undetermined value.

After the value in *ptr* was allocated and freed during function execution, *ptr* is set back to zero. Although this might be considered unnecessary, it is good coding practice to remove the value from pointers that are no longer pointing to a system resource. Hence this coding style is very much in line with a well-organized housekeeping strategy, even if *ptr* is never used later on.

The other reason to reset *ptr* after being freed is, of course, to avoid a forbidden *free( )* in the **ERR\_CATCH { }** section, in case another exception is thrown later in the program flow.

There are some improvements concerning the double implementation of the *free()* function in the standard code and the **CATCH { }** section. Refer to [1.5.2 “ERR\\_TRY, ERR\\_THROW, ERR\\_CATCH” on page 1-20](#) to learn more about this technique.

## 1.6 Coding strategy

The detailed layout of transitions is at the discretion of the programmer

Be sure to change the default font from Arial to Helvetica

Select the / line button to edit a curvy transition

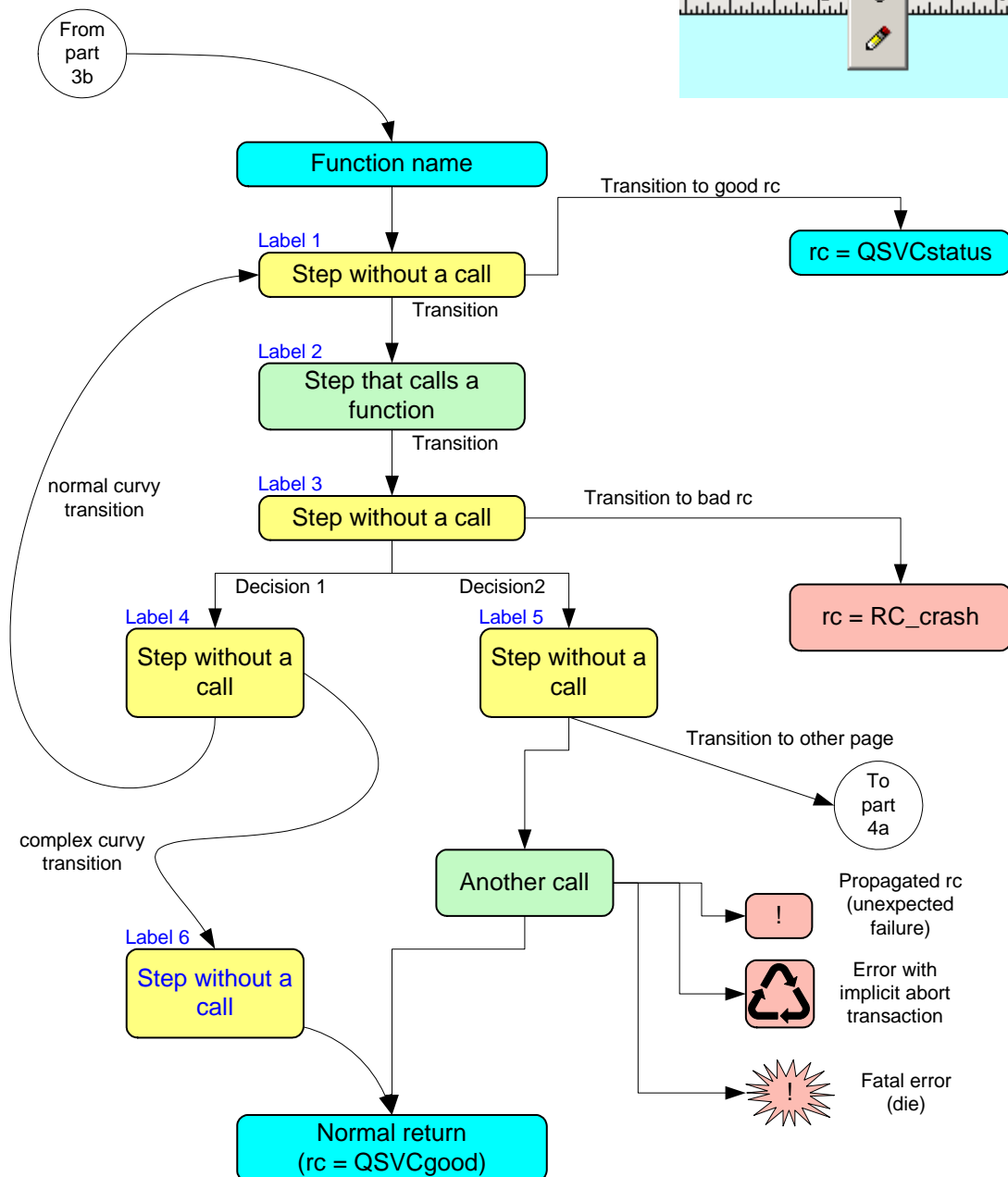
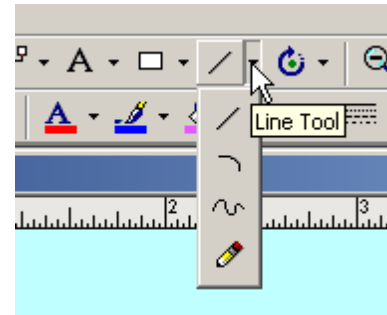


Figure 1-1. State diagram

Figure 1-1 shows a state diagram template that represents the high level design of a module. We put a particular focus on the “Label n” (n = 1 ... ) notations.



**Author's Note:** The above diagram is still a helpful tool. However, if UML tools (Enterprise Architect) are available, then UML activity diagrams might replace the state diagram from Figure 1-1.

As a general strategy, the particular sections or steps in the high level design should correspond to an appropriate entry in the low level design, i.e. it should match a similarly named **@section** or **@step** tag in the ASCII source. This strategy allows an efficient reference from and to the high level design of a code piece.

Furthermore extensive use of **<xref ....>** cross references should be made. Those cross references preferably point to the **functional specification** in order to allow follow up of the implemented coded in response to the appropriate chapter/section of a specification.

---

## 1.6.1 Section strategy

Within the function flow, each functional step shall be characterized by an appropriate section. There is a tag available to mark these sections. The tag shall be unique, however it does not need to assign a meaningful abbreviation. Its purpose is to **serve as an identifier in a state diagram** (see 1.6above).

The following example demonstrates this.

```
//-----  
// @section LtaE | Loop through all entries | All entries are visited  
//           in the Sfi list in order to evaluate the matching  
//           parameters  
//-----  
  
while(*SfiOffset < SfiListSize)  
{  
    rc = PSM_ReadObject(FS_PS->SfiMemId, // PSM_OBJECT_HANDLE handle,  
                        *SfiOffset,      // unsigned int offset,  
                        4,                // unsigned int length,  
                        &SfiEntry);      // unsigned short pBuffer);  
  
    if(rc)  
        ERR_THROW(rc);  
  
    //-----  
    // @cont If the <v Sfid> field in <v SfiEntry> is found to be  
    //       zero, the end of the list is reached. The loop will be  
    //       closed in this case. <v SeekMemId> is, however, still  
    //       zero, to indicate that no entry was found during the  
    //       search.  
  
    rc = rcXX_SfiNotFound;  
    if(!SfiEntry.Sfid)  
        break;  
  
    //-----  
    // @cont If the entries in the <v>SfiEntry<ef> match the input  
    //       variables <v>DfMemId<ef> and <v>Sfi<ef>, then  
    //       <v>SeekMemId<ef> is set to the corresponding target  
    //       <sv MemId> stored in <v>SfiEntry<ef> and the search loop  
    //       is closed.  
  
    if(((SfiEntry.DfMemId & 0x0FFF) == DfMemId) &&  
        (Sfi == SfiEntry.Sfid))  
    {  
        *SeekMemId = SfiEntry.TgtMemIdLow + ((SfiEntry.DfMemId >> 4) & 0x0F00);  
        rc = 0;  
    }  
}
```

```

    break;
}

//-----
// @section Fnrc | Find next record | The next record is found by...

```

The example demonstrates the flavor of documentation for the section body. Each logical unit is described by an **@section** tag with an *identifier*, a *short description*, and a *longer description*. These three fields are separated by a vertical bar.

The **@section identifier** can be a number, but also any other text. It is recommended to keep the text short, because its main purpose is to create a (unique) identifier that can be easily used to find a particular part in the source. This *identifier* must be unique within the function.

The **@section short description** should be used in order to have a small title for the particular coding step that follows. This is to help a reader to see the following code as a functional unit to accomplish a task until the next **@section** is found.

The **@section longer description** is used to explain more details of the code that follows. It may convey some implementation ideas to help the reader understanding the strategy of the implementation of the next block.

If a logical block has some functional sub-blocks, they shall be preceded by a **@cont** tag. This tag makes it clear that it belongs to the same logical **@section** unit, however, it is described as its own functional (sub)unit.

In particular, these steps may be referred to by the low-level implementation documentation. However, it is highly recommended to put as much explanation as possible into the actual code. Other programmers may then use GREP utilities and reasonable ASCII editors to easily understand the purpose and interface of a function or parts of it.

## 1.6.2

### Step strategy

The 'Step strategy' has exactly the same idea as the 'Section strategy' as described in the previous paragraph. The only difference comes from the fact that the abbreviated term is replaced by a number.

- The advantage of '**step #**' is, that someone can easier find himself in the code since the steps are in enumerated order.
- The advantage of '**section <name>**' is that the names can easier be recognized in a **state diagram** (see Figure 1-1) if the **section name** matches **label**.

It is a personal preference which of the ideas to use since they have quite the same quality and effect.

The following example demonstrates the step strategy.

```

//-----
// @step 3 | Loop through all entries | All entries are visited in
//          the SFI list in order to evaluate the matching
//          parameters
//-----

```

A # sign allows post processing tools to automatically replace the # sign by the correct step number. If any other value is written instead of the # sign, then this number will be constant and not altered depending on the invocation options of the appropriate post-processing tool.

Automatic numbering restarts with a base of '1' on each occurrence of a @func tag.

**Note:** Of course, using automatic step numbering comes at the price of losing the intuitive view in the actual source code. It is recommended to replace # representations when a module is considered to be final.

---

### 1.6.2.1

#### Rationale

Using the **@section (@step)** scheme, visiting code modules of other programmers is easier. In particular, without it, the evaluators will have to visit the code with the low-level design at their side. It takes the human brain large amounts of effort of repeated comparisons to understand which statement in the low-level design describes which particular part in the code.

Using the **@section**-commenting syntax is a great relief for the recognition mechanism in human brain. The text in the low-level design can be read more confidently and assigned to the actual part in the code that is the target of the description.

The **@section**-coding style does, however, have another positive impact on the readability of the code. By putting as much text as possible from the low-level design into the **@section** and **@cont** sections, much of the code can be read more intuitively just by reading the source code itself.

This is particularly effective if a programmer, in a debugging session, steps through the code, which might not even be his target of investigation, because he is tracing an error that includes the execution of the easier it is for a programmer to follow the idea of the code design. This not only pays off with the evaluators, but also with any programmer, unless he is reading his 'own' piece of code, where programmers mostly do not need any comment (except when they visit their code 6 months later).

It is unacceptable to have such a programmer (in a debugging session) to be required to find the appropriate part in the low-level design in order to understand the steps of a module. Neither would an evaluator want to do this. Placing low-level design into the actual code helps. The **@section**-code style is a minimum effort that should be established to solve these problems.

Last but not least, there is an option to generate low-level design code from the source, which could, in addition, solve the problem of synchronizing documentation with the actual code. The application of those tools is available and well tested, but it is not described in this document.

---

## 1.6.3

### Indenting comments

Comments shall be indented in-line with the code. It is known that this is at the cost of some extra lines for a comment because there are fewer characters available until column 80.

---

## 1.6.4

### Function declarations

Pointer arguments in function declarations shall either incorporate explicit space qualifiers (e.g., near idata) or shall have a type that is a typedef that incorporates explicit space qualifiers.

Since we now have the parameter descriptions in the header as **@parm's**, the following is **acceptable**:

```
void s_long_exp(WORD M_addr, BYTE M_length, WORD Nn_addr,
               BYTE Nn_length, WORD e_addr, BYTE e_length,
               BYTE Cn_addr, BYTE options);
```

and the following is no longer necessary:

```
void s_long_exp(WORD M_addr,      // pointer to message M
               BYTE M_length,    // length of M
               WORD Nn_addr,     // pointer to norm.modulus Nn
               BYTE Nn_length,   // length of norm.modulus Nn
               WORD e_addr,      // pointer to exp in RAM
               BYTE e_length,    // length of exponent in bytes
               BYTE Cn_addr,     // pointer to result
               BYTE options);    // RAM/EEProm indicators
```

---

## 1.6.5 Indent levels and braces

The indentation step size shall be 2 columns.

**Indentation shall be strictly followed.** An indentation shall always begin **after** an opening brace and end **before** a closing brace.

**No statement shall follow a brace on the same line.** (Line-comments are permitted.)

A brace shall not follow a statement in the same line

All instances of **if**, **for**, **while**, etc. shall be followed by braces, with exception of the “**else-if**” construct, when used during a sequence of tests on a variable, as noted below.

The following are **unacceptable**:

```
void Test(int a) {           // A brace shall not follow a statement.
    a *= 2; }                // A brace shall not follow a statement.

void Test(int a)
{ a*=2; }                    // A statement shall not follow a
                             // brace!

void Test(int a)
{                             // No indention after a brace
    a *= 2;
}

void Test(int a)
{                             // Indent level > 2!
    a *= 2;
}

void Test(int a)
{                             // Opening brace not in same column
    a *= 2;                  // as closing brace
}
```

The only **accepted** declaration is:

```
void Test(int a)
{
    a *= 2;
}
```

In particular, the first **unacceptable** example “if (a == b) {” is generated by many auto-formatting ASCII editors, but is **unacceptable** in this project to avoid confusion with missing braces.

The same rules apply to:

```
while( ... )
{
    ...
}

for(idx = 0; idx < max; idx++)
{
    ...
}

do
{
    ...
} while(!ended);

if(a == b)
{
    ...
}
else if(a < b) // highly recommended to avoid unnecessary indenting
{
    ...
}
else
{
    ...
}

switch(idx)
{
    case 1:
    {
        ...
        break;
    }

    default:
    {
        ...
    }
}
```

---

## 1.6.6 Multiple Statements Per Line

In order to allow efficient debugging with debug tools, each statement shall be placed on its own line. Thus the following construct is **unacceptable**:

```
if (a == 0) b += 2;
```

since the programmer cannot set a breakpoint at the “b += 2”.

Instead, this shall be written as:

```
if (a == 0)
{
    b += 2;
}
```

(... which is still bad practice because the names are single letters.)

Also, do not place the else clause behind the brace.

The following is **unacceptable**:

```
if(a < b)
{
    a *= 2;
    b /= 2;
} else    // this else should be on its own line
{
    ...
}
```

The reason for this isn't very obvious. However it has been shown, that the human brain does recognize the flavor of the **else** part much better if the **else** is actually preceding the part that it owns rather than following the part that it has nothing to do with.

An **acceptable** construct is when an **else** is followed by an **if ( )** when doing a sequence of tests.

Another **unallowed** situation is the nesting of statements

```
if (RC_ErrorFound == (rc = MyFunc(MyParm)))
{
    ERR_THROW(rc)
}
.....
```

is not allowed. The reason is, because during debugging a breakpoint cannot be set on the invocation of *MyFunc*. This is an inconvenience which shall be avoided.

---

## 1.6.7 ‘goto’ statements

**goto** statements are generally not allowed in ‘C’ code. However, there is an implicit exception from this which is, however, not actually coded as a **goto** statement. This is the use of the virtual exceptions as described in sections [1.5.2 “ERR\\_TRY, ERR\\_THROW, ERR\\_CATCH”](#) on page 1-20.

---

### 1.6.7.1 Rationale

The use of **goto** statements has been proven to produce many conflicts in understanding and is considered bad practice throughout the ‘C’ community. Regardless of the fact that they may sometimes be effective, **goto** statements shall not be used anywhere. It has been demonstrated that the same effect can always be achieved with a minimum of additional effort, while the readability does significantly improve without the use of **goto**'s.

---

## 1.6.8

### Code optimization

The purpose of writing code in 'C' is not primarily the size optimization of the compiled object code. In particular, **readability** has highest priority. Therefore, a coding style that is more readable shall, in general, be preferred to a code that sacrifices readability for any kind of optimization.

On the other hand, readable 'C' code can be written that is also size/performance effective.

Size optimization in 'C' is mostly achieved when splitting a large function into subfunctions that may also be called from elsewhere. This practice is recommended.

Performance optimization can mostly be made at the design level, hence this document cannot give any recommendations.

It is **unacceptable** to use **goto** statements in order to save a few bytes in the compiled code, because it would sacrifice a general concept (no **goto** statements) for the sake of a few bytes of optimization.

---

## 1.6.9

### Comments

**Note:** Since there is an intentional freedom in the design of comments, if you edit an existing file, then please don't apply your own style, but adhere to the style used in the file. This is to avoid code that appears differently within a single module. Coding standard violations, however, are of course subject to change.

**Position:** Comments in 'C' typically start in column 1 or at some tab position. Comments shall be aligned in order to provide an easy-to-read layout.

**Layout:** Comments shall not extend beyond column **120**, as this generates undesired line breaks in code printouts. In order to avoid lines longer than 120 chars, it is wise not to extend 80 chars in general. Often during indentation of comments, blocks move to the right and might reach the limit of 120 chars, in order to avoid re-editing, therefore the coder should plan this contingency by not exceeding 80 chars on average.

The rationale for this rule is the readability aspect. Typically column 80 allows plenty of space to write proper code.

**Code sections:** Larger sections of code, particularly function headers, shall be separated by eye-catching header blocks, such as:

```
↓ Column 1                               Column 80 ↓
//=====
//
// void Test(int lines)
//
// Function: does nothing with its input
//
// Input:    lines = number of lines in text
//
// Output:   none
//
//=====
void Test(int lines)
{
```

Smaller sections shall be indicated with small blocks, such as:

```
//-----
```

```
// calculate u*x mod b, then rotate z to get the m of v into 14.
```

as appropriate.

**Style:** The “`/**` **comment**” is highly recommended and preferred over the “`/* ... */`” variant since it is easier to maintain.

**Content:** Comment text should link the code to the problem to be solved; it should **not** comment what a statement **does**. However, it is often useful to indicate the current **content** of a variable.

In ‘C’ code, the focus of comments should establish a link to the state diagrams of the *low-level design*, if available.. This is described in more detail in [1.7.7 “Section strategy” on page 1-41](#).

**Continuation:** Code lines with comments that overflow shall continue on the next line, starting in the same column as the comment on the code line, i.e., the comment delimiters shall be aligned.

**Comment boxes** shall end with a comment line similar to the comment line beginning the comment block.

There shall be no comment lines consisting of all slashes or characters other than mentioned above.

**Alignment:** Long-width comment blocks shall be **code aligned**, such as:



```

rc = RC_SfiNotFound_6984;
if (!SfiEntry.Sfid)
{
    //////////////////////////////////////
    // @cont
    // @xmp
    // .<---- r1l ---->. r0l                                r0l + r0h
    // +-----+-----+-----+-----+
    // |               .               |
    // +-----+-----+-----+
    //               +-----+-----+-----+
    //               |               .               |
    //               +-----+-----+-----+
    //               .<---- r1l ---->. RPTR3 (target addr)
    // @exmp

    SetPtr(RPTR3);

```

This is especially important when pseudo-code of the form “*value* = &*DriverBase*[*MAX\_DRV*]” is present in the *low-level design*.

In *Assembler code*, the focus of comments should concentrate on the **content of registers**. For subsequent code reviews and maintenance, it is essentially important to have an awareness of the current register values when the code is executed.

In particular, the statements of the *low-level design* should be found in the implementation as comments.

**Note:** The actual content of comments is always a matter of personal preference and experience of the programmer, and is difficult to specify in more detail. As a general rule, however, it should be assumed that a person, reading the code, is seeing it for the first time and requires some help to understand what’s going on.

There shall be no comment lines consisting of all slashes or characters other than mentioned above.

**CrossReferencing:** The use of the <xref *crossref*> tag (see 1.4.4 “Other tags” on page 1-14) can be essentially useful for a reader after processing the code with the AUTODOC tool. In particular, if the cross reference points to the specification which described the coded function in a more general view.

## 1.6.10

### Naming convention

Naming convention applies to variables and function names, for constant names see the next chapter 1.6.10.4 “Constants”.

Naming convention **has made history** throughout programming.

#### EXTRACTFILENAME

was the early BASIC approach

#### extractfilename( )

was the corresponding ‘C’ notation. This style can still be found in the function prototypes of standard ‘C’. (memcpy, memcp, malloc etc....)

#### extract\_file\_name( )

improved the ‘C’ style in the 90’s.

#### ExtractFileName( )

is the current common practice used in ‘C’, ‘C++’ and JAVA.

Sometimes a combination of the methods seems to be appropriate

### **fsExtractFileName( )**

is reasonable to identify a cluster of functions that belongs to a particular set of functionality, here the file system.

The **prefixBiCapitalization** standard has been proven optimal regarding the way human brain percepts and memorizes the name of a function. The prefix allows a fast GREG of functions belonging to a typical functionality while the BiCapitalization has been proven to be most memorable while the **intuition and recognition is optimal**.

**Bad practice** would yield the following problems:

### **fs\_extract\_file\_name ( )**

seems to be intuitive too. However, in more complicated situations, the **human brain invents derivatives** when memorizing it, e.g., **fs\_extract\_filename ( )**.

The major disadvantage comes **with the usage of underscores**. In projects that have hundreds of variable names, it is often **not intuitive** where to put or expect an underscore.

Although **GREG** tools allow wild cards to find them, this problem appears to be significant for the easy of reading code. Even using GREG tools, the brain needs some phantasy, where it is possible that an underscore might have to be expected in order to define the search mask.

Therefore the **prefixBiCapitalization** standard shall be used for external functions in O/S kernel.

The names of functions/labels shall be

- significant (**ExtractFileName** is better than **CutFn1**)
- easy to memorize (**ExtractFileName** is better than **ExFileN**)
- easy to type (**ExtractFileName** is better than **ExtrctFn**)

This is essentially important in debugging sessions.

---

## **1.6.10.1**

### **NameLength**

The recommended length of names ranges from 4 to 12 characters.

Single and double letter names should be avoided in any case, as it is difficult to retrieve them with a **GREG** utility.

**Bad practice:**  $mFL = mFh + mFds;$

**Bad practice:**  $TheLengthOfTheFileIncludingHeaderAndDataPart = ..... + ....$

**Good practice:**  $FileLength = FileHeader + FileDataSize;$

**Note:** Although the 'bad practice' example do slightly exaggerate, those styles have already been found in extreme situations. The idea of these 'extreme' examples is to give the reader a clear idea about the issue being discussed, while allowing the personal freedom that programmers need in order to work effectively.

---

## **1.6.10.2**

### **BiCapitalization**

Names shall be lower case, except the first letter and some meaningful separator letters. "**PostDes**" or "**PostDES**" reads better than "**postdes**". It is highly recommended to use the first of these three variants **PostDes** instead of **PostDES**.

As DES is a well known abbreviation for the Data Encryption Standard we understand the idea to express this with the name using the capital abbreviation **DES** instead of **Des**. But the trouble starts already with **PostDESRoutine**, which decreases the readability significantly compared to **PostDesRoutine** or it breaks the rule of BiCapitalization by **Post-DESroutine**.

Therefore the multiple capitals are only allowed if they are **at the end** of the name (e.g. **PostDES**), however for general consistency it is **highly recommended** to make use of the conservative variant **PostDes**.

---

### 1.6.10.3 Underscores

Underscores in names should generally be avoided. The rationale is given in 1.6.10 “Naming convention”.

---

### 1.6.10.4 **Constants**

The naming convention for constants is different from the naming convention describe above. It has been found practical to distinguish constants in appearance. Therefore the **prefix\_CAPITAL\_UNDERSCORE** approach is taken for constants.

```
#define      fs_MAX_BUFLength      0x1000
```

is a typical constant declaration that uses all capital letters with an underscore to separate the words. The **prefix\_CAPITAL\_UNDERSCORE** syntax is reserved for **constants** exclusively !

As described before the **prefix** is used to specify the module group (e.g. key system, file system) that the constant belongs to.

**Note:** In particular circumstances a programmer might find the use of other constant names reasonable. An example for this would be the definition of return code verbose names or the tokens of a *switch( )* statement. Therefore the use of this naming convention should be reasonably taken for situations where configuration constants (like buffer size, flag parameters etc) are being specified and exported.

---

## 1.6.11 **Header local defines**

Each header file shall begin with a unique name definition in order to avoid cyclic or repeated inclusions of the same header file.

This is a common practice in ‘C’ code anyway. However, the following naming convention shall be used.

A file **filename.h** shall have the following layout

```
#ifndef _FILENAME_H_
#define _FILENAME_H_

... all sort of things....

#endif
```

The naming convention of the first define shall always reflect the filename and type separated, preceded, and followed by underscores and shall be uppercase.

---

**1.6.11.1****Rationale**

Again, this is a common 'C' practice. This naming convention has the advantage that, with certain editors, one can quickly GREP for a filename (in our example, "\_FILENAME\_H\_" to get to the header file, which might be in different directory. Current processing speed allows such behaviour to be much faster than any search in a directory tree. Of course, this is particularly helpful for ASCII editors that combine a set of basic tools in an integrated development environment (IDE) (e.g., CodeWright is famous for these features).

---

**1.6.12****Errors**

Compiler or linker errors shall not be tolerated.

---

## 1.7 ASM specific implementation standards

---

### 1.7.1 Source File Headers

Every Assembler source file will have a header, as shown here. There is an equivalent header for Assembler “.inc” files.

```
*****
;*                                                                    **
;*                                                                    **
;*   GuD Confidential                                                    **
;*                                                                    **
;*   (C) Copyright Giesecke & Devrient Corporation 2001-2006          **
;*                                                                    **
;*   LICENSED MATERIALS - PROPERTY OF GIESECKE & DEVRIENT            **
;*                                                                    **
;*   All Rights Reserved                                                **
;*                                                                    **
;*   Origins:  GDM / NB4 Munich                                         **
;*                                                                    **
;*   *****
;
;
;   @modname      (name of the module)
;
;   @moddesc      (description of the module)
;
;   @status       Version 1 - Release 2
;
;   @restrictions (any restrictions - if none, say “None known”)
;
; Tag  Identifier  Description
; ----
; @rel Hsc06Mar01 | 0 | Initial Version 1 Release 0
; @crv Asw06Mar02 | Hack, LePuce, ...
; *****
; @doc
; @decl
```

These tags and methods are similar to [1.4.3 “Function headers” on page 1-12](#), [1.4.1 “Source File Headers” on page 1-11](#)

---

### 1.7.2 Include strategy

The actual syntax of include statements is compiler/assembler specific. Therefore only recommendations can be given here.

Include statements are typically be of one form, like:

```
$include (processor.inc)
```

Header files shall only be placed into the *include* directory or the component's own directory. Due to code modularization constraints, modules shall not “include” header files from other modules.

---

### 1.7.3 External Function Names

Every external function *MyFunction* often might have three forms of the name:

- the plain “un-decorated” name, e.g. *MyFunction*
- the name with a leading question mark, e.g. *?MyFunction*

- the name with a leading underscore, e.g. `_MyFunction`

This is a compiler specific feature and not required in the general case.

## 1.7.4 Function Header

A 'C' code function has an encapsulated structure:

```
FunctionName(type A, type B)
{
    .... code ....
}
```

In Assembler, this type of encapsulation is often undesirable when minimizing code size. This is particularly applicable to SmartCards. For the purpose of dense code, it is sometimes reasonable to jump into the body of another function.

As a response to this dilemma, the different parts of Assembler code have been designed as modular units, although they are not implemented as called functions. The reader, however, might regard them as being similar to inline functions in 'C'.

There is a strict convention that each of these routines has its own header. (A routine is defined as beginning wherever there is a label that is called from or jumped to from another routine.) These headers shall have a generic layout that allows a structured analysis of the function. The generic structure and some additional tags are required to make the function headers machine readable. This is particular important for *automated code analysis*.

Labels used in only one place do not require their own header; for instance, a loop entry-point does not require a separate header, as it is called only once in the module. If, however, a label (even a loop entry point) is called from elsewhere, it will require a header in order to display the interrelationship to its callers.

A typical routine header consists of the following entries:

```

↓ Column 1                                                    Column 80 ↓
;=====
; @func void | SetOne | puts the value 'one' to the specified address
;                               with a length of r0h
;
;
; @parm UCHAR      | r0h | length (mention reg and stack var)
; @parm FMXADDR_P | r0l | start addr
;
; @output
;     first byte of (little-endian) byte pointed to by <start addr>
;     is set to '01'x. All other bytes are reset to '00'x
;
; @desc The function considers the memory area addressed by
;       r0l as a little-endian number with length r0h. The
;       value of this 'variable' is set to one, i.e. the all bytes
;       from the start address r0l to a length of r0h are set to
;       zero and the first (least significant byte) of the variable
;       is set to 0x01.
;=====
```

The function declaration, as shown here, allows a standardized, machine-readable way of defining functions. In the following, a short rationale is given for each of the parts of the declaration.

- @func** This tag specifies the function type, name, and short description. The short description shall be a line text that would fit into one line of a typical table of contents. A more detailed description shall be given in the **@desc** section later.
- A vertical bar shall separate the fields.
- This tag is important to allow evaluation programs to identify a function declaration. The fields shall to be separated by the vertical bar ('|') character to allow later interpretation of the function declaration by a post-processing program.
- Note:** A way to find a function implementation is to GREP on "`| FuncName`". Since the bar with the name occurs only within the declaration **@func**, the function implementation may be found quickly. Many specialized ASCII editors (e.g., CodeWright) allow effective '**grepping**' through all project files on a GUI level.
- @parm** This tag specifies a function parameter type, name, and description. The description shall explain the use and meaning of the parameter. It is highly recommended to describe the parameters with an "On entry,... On return,..." context in order to clearly identify what is expected when the function is called and/or when the function returns, such as:
- ```

; @parm *int | Index | On entry, Index contains the current
;           current index into the table; on return, this value
;           is incremented by one.

```
- Due to the different nature of parameters, however, this cannot be enforced.
- A vertical bar shall separate the fields.
- There shall be as many **@parm** tags as parameters in the function. **@parm** shall be used only if the function has parameters.
- These tags are important for evaluation programs to learn about the function's parameters.
- @output** This tag precedes the explanation of the function's return codes.
- @output** shall be used always, because a reader always wants to know the result of a function.
- @rc** These tags list the return codes that are generated by this function. Return codes that are propagated from called subfunctions shall not be listed. This method makes it easier to maintain consistency between the actual code and its return code description.
- A vertical bar shall separate the fields.
- @rc** shall be used if the function returns return codes. However, return codes, that the function propagates from called subfunctions, shall not be listed with **@rc** tags. Hence, whenever a function returns a return code that originates in a called subfunction, a **@rc** tag is not included in the description of the function header, even though it returns a return code.
- @desc** This tag lists the function description. A detailed idea of the function's flow shall be included here.
- The tag is important to separate the descriptive part from the machine-readable sections.
- The use of **@desc** is recommended, if reasonable.

### @pseudocode

This tag invites a pseudo-code section. It is optional and should be used only if a pseudo-code section is present.

The tags being used in the function headers match those in syntax and meaning which are described in 1.4.3 “Function headers” on page 1-12. A particular advantage of these tags is, that, regardless of the type of source (ASM or ‘C’) the function headers match and allow a reader an intuitive perception of the required information to understand a function.

---

## 1.7.5 Other tags

Other tags that may be used throughout to indicate special comments.

### <v varname>

The example above also shows some special tags. These tags may optionally be used to switch fonts if auto-documentation tools are used during post-processing. Their application is not mandatory, but may serve special purposes. The specification of these optional tags may be found in other parts of this documentation.

### @dpa

This tag identifies the areas for DPA comments

### @edpa

Since a DPA comment usually pertains to a section of code that may be sensitive to a DPA attack, this tag identifies the end of the sensitive code.

### @destroys

comma separated list of names which are scratched within a routine  
Syntax: **@destroys** RecNo, pUpdate, sLink .....

### @globalv

followed by a type definition, a global variable and a description  
Syntax: **@globalv** <type> | <varname> | <description>

### @xmp

This tag identifies the area for a code example within a comment

### @exmp

This tag identifies the end of the code example within a comment

Example:

```
-----  
; @xmp  
; If MSB of ExpWord == 1 we set  
;  
; XPTR2 = RPTR2 - input for X**2 is output of multiply  
; RPTR2 = XPTR2 - output for X**2 is input of multiply  
;  
; else (MSB of ExpWord == 0), we ignore the multiply  
;  
; XPTR2 = XPTR2 - input for X**2 is input of multiply  
; RPTR2 = RPTR2 - output for X**2 is output of multiply  
;  
; always do ....  
;  
; XPTR1 = RPTR2  
; RPTR1 = XPTR2  
;  
; The above algorithm can be reduced to  
;  
; if bit == 1  
; XCH(XPTR2,RPTR2)  
; XPTR1 = RPTR2  
; RPTR1 = XPTR2  
; @exmp  
-----
```



## 1.7.6

## Unique return codes

see 1.5.1 “Unique return codes” on page 1-17

## 1.7.7

## Section strategy

Within the function flow, each functional step shall be characterized by an appropriate section. There is a tag available to mark these sections.

The following example demonstrates this.

```
;-----  
; @section cTg | Check for T=0 | Check if a TD1 byte is present  
;               by looking in T0; if it's present, count the  
;               bits in T0 to determine offset to TD1.  
;-----  
movc.b  r3l,[r2+]      ; fetch ATR's T0 byte  
rlc.b   r3l,#1         ; do we have a TD1 (Carry)?  
bnc     SendATRcont    ; nope, so assume T=0  
;-----  
; @cont We have a TD1; rotate T0 and count the bits that are on.  
;-----  
  
movs.b  r3h,#3         ; set loop count  
movs.w  r4,#0         ; set byte count  
  
ATR_Tloop:  
rlc.b   r3l,#1         ; if bit is on,  
addc.w  r4,#0         ; add 1 to the byte count  
djnz    r3h,ATR_Tloop  
  
;-----  
; @cont Fetch TD1 and test for T=0 and T=1  
;-----  
  
add.w   r2,r4          ; point to TD1  
movc.b  r3l,[r2+]      ; fetch TD1  
and.b   r3l,#0Fh       ; just the protocol bits, please  
bz      SendATRcont    ; if 0, it's T=0  
cjne    r3l,#1,ATR_protBad ; ATR Protocol not T=0 or T=1  
  
;-----  
; @section Dmo | Do more | .....  
;-----
```

The example demonstrates the flavor of documentation for the section body. Each logical unit is described by an **@section** tag with an *identifier*, a *short description*, and a *longer description*. These three fields are separated by a vertical bar.

The **@section identifier** can be a number, but also any other text. It is recommended to keep the text short, because its main purpose is to create a (unique) identifier that can be easily used to find a particular part in the source. This *identifier* must be unique within the function.

The **@section short description** should be used in order to have a small title for the particular coding step that follows. This is to help a reader to see the following code as a functional unit to accomplish a task until the next **@section** is found.

The **@section longer description** is used to explain more details of the code that follows. It may convey some implementation ideas to help the reader understanding the strategy of the implementation of the next block.

If a logical block has some functional sub-blocks, they shall be preceded by a **@cont** tag. This tag makes it clear that it belongs to the same logical **@section** unit, however, it is described as its own functional (sub)unit.

In particular, these steps may be referred to by the low-level implementation documentation. However, it is highly recommended to put as much explanation as possible into the actual code. Other programmers may then use GREP utilities and reasonable ASCII editors to easily understand the purpose and interface of a function or parts of it.

---

### 1.7.7.1

#### Step strategy

The 'Step strategy' has exactly the same idea as the 'Section strategy' as described in the previous paragraph. The only difference comes from the fact that the abbreviated term is replaced by a number. The advantage of '**step #**' is, that someone can easier find himself in the code since the steps are in enumerated order. The advantage of 'section <name>' is that the names can be easier recognized in a state diagram as tagged to the states. It is a personal preference which of the ideas to use since they have quite the same quality and effect.

The following example demonstrates the step strategy.

```
;-----  
; @step # | Loop through all entries | All entries are visited in  
;         | the SFI list in order to evaluate the matching  
;         | parameters  
;-----
```

---

### 1.7.7.2

#### Rationale

Using the **@section** scheme, visiting code modules of other programmers is easier. In particular, without it, the evaluators will have to visit the code with the low-level design at their side. It takes the human brain large amounts of effort of repeated comparisons to understand which statement in the low-level design describes which particular part in the code.

Using the **@section**-commenting syntax is a great relief for the recognition mechanism in human brain. The text in the low-level design can be read more confidently and assigned to the actual part in the code that is the target of the description.

The **@section**-coding style does, however, have another positive impact on the readability of the code. By putting as much text as possible from the low-level design into the **@section** and **@cont** sections, much of the code can be read more intuitively just by reading the source code itself.

This is particularly effective if a programmer, in a debugging session, steps through the code, which might not even be his target of investigation, because he is tracing an error that includes the execution of the easier it is for a programmer to follow the idea of the code design. This not only pays off with the evaluators, but also with any programmer, unless he is reading his 'own' piece of code, where programmers mostly do not need any comment (except when they visit their code 6 months later).

It is unacceptable to have such a programmer (in a debugging session) to be required to find the appropriate part in the low-level design in order to understand the steps of a module. Neither would an evaluator want to do this. Placing low-level design into the actual code helps. The **@section**-code style is a minimum effort that should be established to solve these problems.

Last but not least, there is an option to generate low-level design code from the source, which could, in addition, solve the problem of synchronizing documentation with the actual code. The application of those tools is available and well tested, but it is not described in this document.

## 1.7.8

### Function declarations

Pointer arguments in function declarations shall either incorporate explicit space qualifiers (e.g., near idata) or shall have a type that is a typedef that incorporates explicit space qualifiers.

## 1.7.9

### Function calling conventions

PUBLIC assembler functions should have a 'C' calling convention in order to allow their usage from 'C' code.

## 1.7.10

### Comments

**Position:** Comments typically start a column 1, 9, or 41. Exceptions from this general rule are permitted as appropriate for improved readability.

**Layout:** Comments shall not extend beyond column 120.

Larger sections of code, particularly routine headers, shall be separated by eye-catching header blocks, such as:

```
↓ Column 1                               Column 80 ↓
;*****
;
; Driver installation
;
;*****
```

Smaller sections shall be indicated with small blocks, such as:

```
↓ Column 9                               Column 80 ↓
;-----
; calculate u*x mod b, then rotate z to get the m of v into 14.
;-----
```

as appropriate.

**Content:** Comment text should link the code to the problem to be solved. They should **not** comment what the opcode **does**. It is often useful to indicate the current **content** of a register.

```
mov.w    r0,DriverBase      ; load r0 with Driver Base
add.w    r0,#MAX_DRV        ; add MAX_DRV to r0
mov.w    value,r0           ; store r0 into 'value'
```

is **unacceptable**. At least use:

```
mov.w    r0,DriverBase      ; r0 : DriverBase
add.w    r0,#MAX_DRV        ; r0 : DriverBase+MAX_DRV
mov.w    value,r0           ; r0 -> value
```

or even better:

```
mov.w    r0,DriverBase      ; value = &DriverBase[MAX_DRV]
add.w    r0,#MAX_DRV        ;
mov.w    value,r0           ;
```

or even better yet:

```
-----  
; value = &DriverBase[MAX_DRV]  
-----  
mov.w    r0,DriverBase      ; r0 : DriverBase  
add.w    r0,#MAX_DRV        ; r0 : DriverBase+MAX_DRV  
mov.w    value,r0           ; r0 -> value
```

**Continuation:** Code lines with comments that overflow shall continue on the next line, starting in the same column as the comment on the code line, i.e., the comment delimiters shall be aligned.

**Alignment:** **Short-width** comment blocks shall be **code-line comment aligned**, such as:

```
add.b    r0l,r1l           ; source_addr += number of DWORDs of overlap  
sub.b    r0h,r1l           ; length      -= number of DWORDs of overlap  
add.b    RPTR3,r1l         ; target_addr += number of DWORDs of overlap
```

```
-----  
;  
;  
; This is a picture of the stack at this  
; time:  
;  
; +-----+  
; |           parm 1           |  
; +-----+  
; |           parm 2           |  
; +-----+  
;  
;  
-----
```

```
dclx:  
    call    FmxDownCpy      ; Copy current chunk of r0h DWORDs
```

**Note:** However, remember, that drawings can also be included with the <Fig Name> statement.

**Long-width** comment blocks shall be **code aligned**, such as:

```
add.b    r0l,r1l           ; source_addr += number of DWORDs of overlap  
sub.b    r0h,r1l           ; length      -= number of DWORDs of overlap  
add.b    RPTR3,r1l         ; target_addr += number of DWORDs of overlap
```

```
-----  
; .<---- r1l ---->. r0l   r0l + r0h  
; +-----+  
; |           .           |  
; +-----+  
;  
; +-----+  
; |           .           |  
; +-----+  
; .<---- r1l ---->. RPTR3 (target addr)  
-----
```

```
dclx:  
    call    FmxDownCpy      ; Copy current chunk of r0h DWORDs
```

This is especially important when pseudo-code of the form “*value = &DriverBase[MAX\_DRV]*” is present in the *low-level design*.

In Assembler code, the focus of comments should concentrate on the *content of registers*. For subsequent code reviews and maintenance, it is essentially important to have an awareness of the current register values when the code is executed.

In particular, the statements of the *low-level design* should be found in the implementation as comments.

The actual content of comments is always a matter of personal preference and experience of the programmer, and is difficult to specify in more detail. As a general rule, however, it should be assumed that a person, reading the code, is seeing it for the first time and requires some help to understand what's going on.

There shall be no comment lines consisting of characters other than mentioned above.

---

## 1.7.11

### Label naming

The names of labels shall be

- significant
- easy to remember and
- easy to type

This is essentially important for debugging sessions.

The recommended length of label names ranges from 4 to 12 characters.

Single and double letter labels should be avoided in any case as it is extremely difficult to retrieve them with a GREP utility.

*Public labels*, called from 'C' shall contain an underscore "\_" as first character. This is a function of the Tasking 'C' compiler, which, for a routine *foo*, actually generates the external label *\_foo*. It shall also contain a similar label with a "?" as the first character. (Ref: [1.7.3 on page 1-37](#)).

---

### 1.7.11.1

#### NameLength

The recommended length of names ranges from 4 to 12 characters.

Single and double letter names should be avoided in any case, as it is difficult to retrieve them with a **GREP** utility.

**Bad practice:**      mFh + mFds;

**Bad practice:**      TheLengthOfTheFileIncludingHeaderAndDataPart1

**Good practice:**    FileLength, FileHeader, FileDataSize;

**Note:** Although the 'bad practice' example do slightly exaggerate, those styles have already been found in extreme situations. The idea of these 'extreme' examples is to give the reader a clear idea about the issue being discussed, while allowing the personal freedom that programmers need in order to work effectively.

---

### 1.7.11.2

#### BiCapitalization

Labels shall be lower case except the first letter and some meaningful separator letters. "**PostDes**" reads better than "**postdes**". When defining an external 'C' label, one underscore as the first character will normally be the only condition where an underscore is allowed.

Examples of [good labels](#) are:

```

_SLongDiv:                ; external 'C' label
PostDes:                  ; significant, 7 letters,
                          ; capital letters in right place

```

Examples of **unacceptable labels** are:

```

DE:                        ; upper case, only two letters
__R0231B3F9:              ; two underscores difficult to read
                          ; non-significant numbers are hard
                          ; to remember
TestRegisterForLeakageAndInvert: ; too long!!!

```

---

## 1.7.12

### Errors

Assembler or linker errors shall not tolerated.

---

## 1.7.13

### Coding Style

As stated above, coding style is frequently a matter of individual preference. Therefore in this paragraph, some general recommendations on coding-style priorities are be outlined:

- Code should be as dense as possible, i.e., a minimum of whitespace.
- Reuse of routines or parts of them is highly recommended.
- If registers are used for long term operations (more than 25 lines of code), depending on the problem, it is recommended that they be assigned a name:

```
define    r11    'LoopCnt'
```

is an example how to assign a name to a register. Subsequently

```
undef    r11
```

removes the definition in order to avoid undesired side effects or confusion with other assignments.

## 1.8 SEQ specific implementation standards

### 1.8.1 Source File Headers

Every sequence file will have a header, as shown here. The preferred comment for sequence files is the semicolon.

```
*****
;*
;*
;*   GuD Confidential
;*
;*   (C) Copyright Giesecke & Devrient Corporation 2001-2006
;*
;*   LICENSED MATERIALS - PROPERTY OF GIESECKE & DEVRIENT
;*
;*   All Rights Reserved
;*
;*   Origins:  GDM / NB4 Munich
;*
*****
;
;
;   @modname      (name of the module)
;
;
;   @moddesc      (description of the module)
;
;
;   @status       Version 1 - Release 2
;
;
;   @restrictions  (any restrictions - if none, say "None known")
;
;
; Tag  Identifier  Description
; ----
; @rel Hsc06Mar01 | 0 | Initial Version 1 Release 0
; @crv Asw06Mar02 | Hack, LePuce, ...
*****
; @doc
```

These tags and methods are similar to [1.4.3 "Function headers"](#) on page 1-12, [1.4.1 "Source File Headers"](#) on page 1-11.

## 1.8.2 Writing test sequences

A typical test sequence uses a number of specific tags to describe a test scenario. The following example demonstrates this.

```
;-----  
; @tcomp BASE  
;  
; @tsubsys Security Environment | Security environments are established and  
;                               tested  
;  
; @tbatch BatchNameExample | This batch collects the test cases that relate to  
;                               volatile storage of a security environment.  
;  
;-----  
; @tcase ScAcc_ArrDF_001 | Test access rule refs tied to a DF, rule is ALW  
;  
; @tresult No output is generated  
;  
; @tdesc This test checks the access to access rules which are set up in a  
;         file EF_RULE. It does not .....  
;  
; @tpre The proprietary BOOT and SET_FCP command must be correctly  
;        implemented to .....  
;  
; @tpost The test case shall erase the EEPROM to clean up  
;  
;-----  
  
;-----  
; @tsection ScAcc_ArrDF_001a | Reset the card | Cold or Warm Reset of the card  
RESET_CARD  
  
;-----  
; @tstep 1 | Create the MF and select it | The <xref BOOT> function initializes  
;         the EEPROM to the default running state. Only the MF will be  
;         available after the BOOT command.  
;-----  
BOOT                ; proprietary command, create MF  
SELECT_ROOT         ; select MF  
  
;-----  
; @tstep 2 | Create Access Rules | We establish the desired access references in  
;         the FCP of the DF. We want to reference the ALW rule in EF_RULE  
;-----  
EF_RULE_reco = '01' ; first record  
SET_FCP('TLV(84, '52 4F 4F 54')); set root name  
.....
```

Figure 1-2. Test sequence example

The used tags have the following meaning:

### @tcomp <chapter title>

describes the component under test. This will generate a head 1 title and shall therefore be used **only once per file** and **only as the first tag**.



The `<chapter title>` designates a name, short enough to fit on one line of the output

### **@tsubsys <title> | <text>**

describes that part of the component, that is tested. Examples are secure messaging, communication layer or access control.

`<title>` is the title of a Head 2 chapter title. `<text>` allows to write some general remark. If no `<text>` is planned, do not forget to set the field separator anyway, otherwise AUTODOC will complain that there are not enough fields.

### **@tbatch <title> | <text>**

describes the name of the actual batch. A test batch is a collection of test cases (see below). This generates a Head 3 section. `<title>` represents the text in the head line.

`<text>` allows to write some general remark. If no `<text>` is planned, do not forget to set the field separator anyway, otherwise AUTODOC will complain that there are not enough fields.

### **@tcase <label> | <text>**

`<label>` is the text for a Head 4 header. **@tcase** describes a test case, more than one test case typically form a test batch. `<text>` describes the test case in more detail.

### **@tresult <text>**

describes the expected result of a test case. If required this can be followed by several **@rc** `<value>` | `<desc>` tags. The **@rc** tag is explained in “**@rc**” on page 1-14.

### **@tdesc <text>**

describes the test case in more detail. Typically this is a longer section that explains the idea and strategy of the text. Several text tags are available in order to achieve a professional layout of this section. Refer to 5 “AUTODOC Tags” in volume 4 page 5-95 for further information.

### **@tpre <text>**

describes the preconditions to be met in order to run the test. Typically these conditions are set by a preceeding sequence, here the name of the sequence can be mentioned. In addition, the essential settings should be noted, i.e. those, which are crucial to understand the mission of the test case.

### **@tpost <text>**

describes the post-conditions that are established after the test. These are important to know, which tests can built upon this sequence, e.g. the card might be defragmented after a test case, another defragmentation test would need additional preparation.

### **@tsection <label> | <title> | <text>**

introduces a section of a test description. Many tests consist of a bring up phase, the test execution and the cleanup phase. These phases can be described in sections. The **@tsection** label will often match an appropriate label in additional design documentation of a test.

### **@tstep <number> | <title> | <text>**

introduces a step of a test description. Many tests consist of a bring up phase, the test execution and the cleanup phase. These phases can be described in sections.

### **@cont <text>**

The typical use of the **@cont** tag is to continue the text of a preceeding **@step** or **@section**. This is necessary when some code is between the **@step/@section** tag and the text to be added. ‘cont’ stands for ‘continuation’.

---

## 1.9

### Final Remark

The coding standards were created to improve and ease communication between programmers in code reviews, on shared code and for maintenance purposes. We believe that a programmer should not become the victim of the coding standards. Therefore these standards try to give programmers as much freedom as reasonable.

The success of these standards will always depend on the personal decision of programmers how close to follow these standards. In this sense your contribution (as a programmer) is highly welcome to support the process of easily interchangeable code.

## 2.1

## About AUTODOC

AUTODOC 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. AUTODOC has traditionally been used to document programming APIs.

Placing API documentation within the source files helps programmers disseminate information about a developing codebase. AUTODOC can generate online Help files containing full hypertext coding with links and keyword lists. Typically, AUTODOC 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.

## 2.2

## AUTODOC Comment Blocks

AUTODOC.EXE scans through a source file and extracts information marked with AUTODOC tags. AUTODOC 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 AUTODOC topic:

```
// CLazyInterface::QueryInterface
//
// @mfunc Implements <om IUnknown.QueryInterface>.
//
// @comm The 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).

## 2.3

## Features of AUTODOC

The following are some interesting features of AUTODOC:

### Flexible Tag Definition

AUTODOC 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. AUTODOC comes with a standard formatting file, AUTODOC.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

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

For more information, see ["Source Parsing"](#).

### Extraction Tags

When performing an AUTODOC build, you can use extraction tags to specify which AUTODOC 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

AUTODOC 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, AUTODOC can create a hypertext link to it. If the topic is not listed, AUTODOC can generate alternate formatting (like bold text)

For more information, see ["Topic Logs"](#).

---

## 2.4

## AUTODOC Tags

AUTODOC 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 AUTODOC tag consists of an ampersand (@) followed by a tag name. Most AUTODOC tags are defined in a format-information file, which is required by the AUTODOC.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. AUTODOC.EXE only recognizes tags that appear as the first item within a line of comment text.

There are three basic types of AUTODOC tags:

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

AUTODOC output combines formatting codes stored in the formatting specification file with field text parsed from the AUTODOC 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, AUTODOC 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 AUTODOC tag encountered in the source file. The @doc signals the beginning of AUTODOC information within a source file and identifies extraction flags, tokens used to classify AUTODOC topics. For example, you can classify AUTODOC 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 AUTODOC 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!
```

---

## 2.4.1 AUTODOC Comment Blocks

AUTODOC-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 AUTODOC comment blocks:

- AUTODOC topics can reside in a single comment block, or they can span multiple comment blocks.
- AUTODOC 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 AUTODOC 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 AUTODOC blocks.

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

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

; Assembly language comment
;
; This text is ignored by AUTODOC.
;
; @doc
```

```

;
; @func int | MyFunc | This function performs a useful task.

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

```

---

## 2.4.2 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.

---

## 2.4.3 Topics Spanning Multiple Comment Blocks

AUTODOC blocks can span several comment blocks. The AUTODOC-tagged text is appended to the preceding AUTODOC 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 AUTODOC; normally, the @FUNC tag requires three fields (return value, function name, and description), so AUTODOC parses the function header immediately following the first comment block to obtain the return value and function name.

---

## 2.5 Source Parsing

AUTODOC 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 AUTODOC 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, AUTODOC 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, AUTODOC will check the formatting-file entry to see whether source parsing is enabled. If it is, AUTODOC will try to extract the missing fields from the source text.

---

### 2.5.1 Supported Source Parsing Configurations

AUTODOC can parse source text from two locations:

A source declaration following an AUTODOC comment block

A source declaration occurring on the same line on which an AUTODOC 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.  
{  
...  
}
```

---

### 2.5.2 Parameter and Structure Field Parsing

AUTODOC 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.



---

### 2.5.3 Function and Member Function Parsing

AUTODOC 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 `".PARSE-SOURCE=function"` or `".PARSESOURCE=memberfunction"` to the tag definition.

---

### 2.5.4 Class Parsing

AUTODOC 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.

---

### 2.5.5 Enumeration Member Parsing

AUTODOC can parse the names of enumeration members.

To enable enumeration member parsing, you must add the statement `".PARSE-SOURCE=emem"` to the tag definition.

---

## 2.6 Nesting Topics Inside Topics

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

---

## 2.6.1 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 AUTODOC'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;
};
```

---

## 2.6.2 Format-File Entries for Nested Topics

To generate the topic tag for a nested topic, AUTODOC 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.



## Using AUTODOC.EXE

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

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

```
AUTODOC [/v] [/e] [/n] [/a] [/u] [/p{+|-}] [/r[dh]] [/t[0-9]]
        [/b filename]
        [/o filename] [/l filename]
        [/f filename] [/c filename]
        [[/x id]...] [[/d name=text]...] files
```

Table 3-1. AUTODOC.EXE Invocation options

Option	Description
[/v]	Prints detailed status information to the console.
[/b filename]	Use reference <filename>[.MIF] to merge figures and tables from this file into the output. Refer to <a href="#">6.4.3 “Step 3: Write the program source(s)” on page 6-143</a> for information how to refer to figs and tbls.
[/e]	Suppresses warnings about empty fields.
[/mIM] <prefix>	Temporarily defines a prefix variable “PreFix=<prefix>” to control the automatic target detection.  This reacts on <xref <b>env</b> _reference> entries and automatically replaces <b>env</b> by the value in the “PreFix” environment variable
[/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.
[/p{+ -}]	Post Process SGML output to remove empty tags. This is the default setting. Using (-) for test purposes disables the post processing.
[/rs]	Generate SGML output for use with the IBM ID Workbench
[/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.
[/i]	parse subdirectories (new feature added for IBM version)
[/o filename]	Use output file <filename>. If no output file is specified, AUTODOC 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.

Table 3-1. AUTODOC.EXEInvocation options

Option	Description
[/f filename]	Use format file <filename>.If no format file is specified, AUTODOC searches for AUTODOC.FMT in the directory where AUTODOC.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 AUTODOC.FMT file.Note that you can also insert your additional entries at the beginning of the AUTODOC.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 AUTODOC.
[/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=c 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.

## 3.1 Makefile Entries for AUTODOC

The DKOALA example project included with AUTODOC uses MAKEDOCS.MAK, a Microsoft Visual C++ makefile. MAKEDOCS.MAK is a generic AUTODOC 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):

```
AUTODOC\$(Project).Hlp
AUTODOC\$(Project).Doc
```

MAKEDOCS.MAK

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

```
# AUTODOC MAKEFILE
#
# Eric Artzt, Program Manager / Helmut Scherzer, change for IBM
# Consumer Division, Kids Software Group
# Internet : erica@microsoft.com
#

OUTDIR = $(ProjDir)\AUTODOC
TARGET = $(Project)
TITLE  = $(TARGET) Help
DOCHDR = $(TARGET) API Reference
AD      = AUTODOC.exe
ADTOC   = "C:\Bin\Contents.D"
ADHLP   = /RH /O$(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)
```

## 3.2 Generating Topic Indexes

AUTODOC 3.0 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.

**Note:** This feature is not necessary for IBM as we generate the TOC right away in the FrameMaker

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
```

### 3.2.1 Default Contents File

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

### 3.2.2 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 AUTODOC 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
```

## 3.3 Conditional Topic and Paragraph Extraction

You can limit the set of topics extracted in an AUTODOC 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).

### 3.3.1 Associating Extraction Tags with Topics

Use the `@DOC` tag to associate extraction tokens with topics. The `@DOC` tag must precede any AUTODOC 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 AUTODOC 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 AUTODOC 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.
```

## 3.3.2 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 AUTODOC command extracts only those topics that have both the `EXTERNAL` and `WIN4J` tags:

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

See “[Extraction and Filtering Expressions](#)” for details on the expression syntax.

## 3.4 Extraction and Filtering Expressions

AUTODOC 3.0 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.

### 3.4.1 Expression Syntax

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

### 3.4.2 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 AUTODOC 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
```

### 3.4.3 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
```

### 3.4.4 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)
```

### 3.4.5 Evaluation Order

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

### 3.4.6 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 AUTODOC command line extracts only those topics with the tag EXTERNAL and any of the tags ONE, TWO, or THREE:

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

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

### 3.4.7 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 expres-  
sion>
```

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
```

## 3.5 Topic Logs

AUTODOC 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, AUTODOC generates RTF formatting for a hypertext link.

**Note:** The RTF and HLP generation was taken out from the IBM format file. It can still be use as described by Eric Arzt, however the new IBM tags will have to be added yet.

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

AUTODOC 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 AUTODOC, 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.

AUTODOC 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 AUTODOC builds, each of which references a central log file naming all the available link targets.

### 3.5.1 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 AUTODOC /C option, as follows:

```
AUTODOC /c mdatopic.log /rh *.cpp *.h
```

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

```
AUTODOC /x EXTERNAL /l newfile.log *.cpp *.h
```

### 3.5.2 Linking Formatting Specs to the Log File

In the formatting specification file, you can specify alternate formatting information for paragraph and text tags. AUTODOC 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 AUTODOC formatting information, you use the tag ".IF" statement, which can be used in formatting blocks for paragraph and text tags and in formatting diagrams.

## **3.6 Defining Tags and RTF Output Strings**

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

### **3.6.1 Locating the Formatting File**

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

### **3.6.2 Adding Supplemental AUTODOC Tags**

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

A supplemental formatting file is organized the same as AUTODOC.FMT. You can copy tags or whole sections from AUTODOC.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 AUTODOC.FMT.

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

## 4

# 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 AUTODOC 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 AUTODOC 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.

## 4.1

### Comments

You can type comments within the formatting file. Preface any comment lines with a semicolon typed at the beginning of the line.

## 4.2

## Format Strings

Format strings consist of literal output text mixed with special entries that reference fields from the AUTODOC 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 AUTODOC 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. AUTODOC 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., Template-Func<class b>), AUTODOC 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.

<b>\$!p</b>	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.
<b>\$!l</b>	Outputs the source-file line number of the tag.
<b>\$!c</b>	Outputs the topic context string.
<b>\$!n</b>	Outputs the topic name.  If referenced in the [FILE] section, this code produces no output.
<b>\$!&lt;number&gt;</b>	Switches the internal paragraph generation mode when two hard breaks are found in the text. <ul style="list-style-type: none"> <li>1 starts paragraph</li> <li>2 generates two hard returns</li> <li>3 generates two new paragraphs</li> <li>4 start with body paragraph</li> <li>5 generate one hard return</li> </ul> Whenever two adjacent CRLF are found in the text while the field hasn't ended, AUTODOC generates the appropriate codes in the source.

## 4.3 [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

[\[CONSTANT\]](#)  
[DEFINE](#)  
[OUTPUT](#)

## 4.4 [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, AUTODOC provides the field information for the paragraph corresponding to the current entry. For the .PRE and .POST items, AUTODOC 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

## 4.5 [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.

### Comments

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

See Also  
[\[EXTENSION\]](#)  
[EXT](#)

## 4.6 [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](#)

## 4.7

# [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.

AUTODOC 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.

**.PRENAME=**  
Specifies a format string to output before each topic name.

**.POSTNAME=**  
Specifies a format string to output after each topic name.

**.PRECONTEXT=**  
Specifies a format string to output before each context string.

**.POSTCONTEXT=**  
Specifies a format string to output after each context string.

### Comments

All format entries are optional.

Index formatting strings can use the field specifiers used for file formatting strings. 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 index
.output=help
.prename=\pard\plain $(h_indexlink){\uldb
.precontext={\v
.postcontext=}\par

Doc index

.output=doc
.prename=\pard\plain $(d_indexlink)
.precontext={\v
.postcontext=}\par
```

See Also  
[Format Strings](#)

## 4.8 [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. AUTODOC 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 AUTODOC 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](#)

## 4.9 [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](#)

## 4.10 [TOKEN] Section

This section defines formatting codes for special characters of a particular output type. Previous versions of AUTODOC 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.

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

### Entries

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

**."TOKEN"=** Defines a control character for the output type.

**."TOKEN"=** Defines a formatting string to use for high-ASCII characters.

### Example

This section defines the standard "doc" and "help" output types provided in AUTODOC 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 \\'x

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

See Also

[Format Strings](#)

[\[TOKEN\]](#)

[HIGHCHARMASK](#)

[OUTPUT](#)

[TOKEN](#)

## 4.11 [TOPIC] Section

This section defines topic tags. These tags identify a single documentation unit, or topic. For example, the standard AUTODOC.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. AUTODOC 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](#)

## 4.12 [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

<i>sName</i>	Specifies the name of the constant. Type any name up to 63 characters, with no embedded spaces, tabs, commas, or semicolons.
<i>sText</i>	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 AUTODOC. Constants defined on the command line override constants with the same name defined in the format file.

### See Also

[\[CONSTANT\]](#)  
[DEFINE](#)  
[OUTPUT](#)

## 4.13 [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

<i>sOutputType</i>	Specifies the output type for the file formatting block. Use one of the following strings:
<i>doc</i>	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.
<i>help</i>	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.



<i>both</i>	Specifies the formatting block is used for both document and help output.
<i>mif</i>	specifies the formatting block being used for FrameMaker output .MIF

## See Also

[\[CONSTANT\]](#)  
[DEFINE](#)  
[OUTPUT](#)

## 4.14 [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

<i>names</i>	Specifies one or more paragraph tag names, with multiple names separated by commas.
--------------	-------------------------------------------------------------------------------------

## See Also

[Format Strings](#)  
[\[DIAGRAM\]](#)  
[CANCELIFPRESENT](#)  
[TAG](#)  
[PARAGRAPH-IF](#)

## 4.15 [DIAGRAM] Section: .TAG Entry

This item defines a new diagram.

[DIAGRAM]  
 .TAG name, sOutputType, sParaType

### Entry Fields

<i>name</i>	Specifies the name of the diagram. Type any name up to 63 characters, with no embedded spaces, tabs, commas, or semicolons.
<i>sOutputType</i>	Specifies the output type for the diagram. Use one of the following strings:
<i>sDoc</i>	Specifies paper (Word document) output. The formatting block with this sOutputType value is used if the user specifies the /RD command-line flag.
<i>help</i>	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.
<i>both</i>	Specifies both document and help output.
<i>mif</i>	specifies the formatting block being used for FrameMaker output .MIF
<i>sParaType</i>	This item defines the name of the repeating paragraph tag used within the diagram.

## See Also

[Format Strings](#)  
[\[DIAGRAM\]](#)  
[CANCELIFPRESENT](#)  
[TAG](#)  
[PARAGRAPH-IF](#)

## 4.16 [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:

<b>C</b>	C or C++ comment style (// or /*)
<b>ASM</b>	Assembly language comments (;)
<b>BAS</b>	Basic comments (')

### 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=nl,c
.ext=d,c
.ext=h,c
.ext=hpp,c
.ext=hxx,c
.ext=asm,asm
.ext=bas,bas
.ext=mst,bas
```

### See Also

```
[EXTENSION]
EXT
```

## 4.17 [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:

<i>doc</i>	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.
<i>help</i>	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.
<i>both</i>	Specifies the formatting block is used for both document and help output.
<i>mif</i>	specifies the formatting block being used for FrameMaker output .MIF

## Comments

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

## See Also

[\[FILE\]  
OUTPUT](#)

## 4.18 [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

<i>sOutputType</i>	Specifies the output type for the index formatting block. Use one of the following strings:
<i>doc</i>	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.
<i>help</i>	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.
<i>both</i>	Specifies the formatting block is used for both document and help output.
<i>mif</i>	specifies the formatting block being used for FrameMaker output .MIF

## Comments

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

## See Also

[\[INDEX\]  
OUTPUT](#)

## 4.19 .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 AUTODOC 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 AUTODOC.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)
```

## 4.20

### [PARAGRAPH] Section: .MAP Entry

Maps fields in the paragraph tags to fields in a topic tag defined within the same AUTODOC 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){\u1db $1}{\v #1} {\u1db $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](#)

## 4.21 [PARAGRAPH] Section: .PARSESOURCE Entry

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

When AUTODOC determines that fields are missing in a tag, it determines whether source parsing is enabled for the tag. If it is enabled, AUTODOC 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:

<i>parameter</i>	Parameter type and name inserted into fields 1 and 2.
<i>field</i>	Field type and name inserted into fields 1 and 2.
<i>enum</i>	Enumeration name inserted into field 1.
<i>classmember</i>	Member type, name, and (if present) parameter list inserted into fields 1 through 3.
<i>bparameter</i>	Parameter passing convention (OptionalByValByRef) inserted into field 1; type and name inserted into fields 2 and 3.

## Example

The following PARSESOURCE item defines parameter parsing:

```
.parsesource=parameter
```

## See Also

[\[PARAGRAPH\]](#)  
[PARAGRAPH-IF](#)  
[MAP](#)  
[PARSESOURCE](#)  
[TAG](#)

## 4.22 [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

<i>sName</i>	Specifies the name of the paragraph. Type any name up to 63 characters, with no embedded spaces, tabs, commas, or semicolons.
<i>sOutputType</i>	Specifies the output type for the paragraph tag. Use one of the following strings:
<i>sDoc</i>	Specifies paper (Word document) output. The formatting block with this sOutputType value is used if the user specifies the /RD command-line flag.
<i>help</i>	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.
<i>both</i>	Specifies both document and help output.
<i>nFields</i>	Specifies the number of fields in the tag. Type a number from 1 to 6.
<i>nNestLevel</i>	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, AUTODOC 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, AUTODOC includes white space (tabs and spaces).

## See Also

[\[PARAGRAPH\]](#)  
[PARAGRAPH-IF](#)  
[MAP](#)  
[PARSESOURCE](#)  
[TAG](#)

## 4.23 [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

<i>sName</i>	Specifies the name of the format tag. Type any name up to 63 characters, with no embedded spaces, tabs, commas, or semicolons.
<i>sOutputType</i>	Specifies the output type for the text tag. Use one of the following strings:
<i>doc</i>	Specifies paper (Word document) output. The formatting block with this <i>sOutputType</i> value is used if the user specifies the /RD command-line flag.
<i>help</i>	Specifies help (WinHelp/Viewer topic file) output. The formatting block with this <i>sOutputType</i> value is used if the user specifies the /RH command-line flag.
<i>both</i>	Specifies both document and help output.
<i>nFields</i>	Specifies the number of fields in the tag. Type a number from 1 to 6.

## See Also

[\[TEXT\]](#)  
[PARAGRAPH-IF](#)  
[TAG](#)

## 4.24 [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  
TOKEN

## 4.25 [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.

## 4.26 [TOKEN] Section: .TOKEN Entry

Defines a control token for the output type, and shows how special characters read from an AUTODOC 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 double carriage return found within an AUTODOC field, or after every line of an AUTODOC 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  
TOKEN

## 4.27

### [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 AUTODOC. 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](#)

## 4.28

### [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](#)

## 4.29 [TOPIC] Section: .PARSESOURCE Entry

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

When AUTODOC determines that fields are missing in a tag, it determines whether source parsing is enabled for the tag. If it is enabled, AUTODOC 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

<i>sParseType</i>	This field specifies one of the following values indicating the type of source parsing:
<i>function</i>	AUTODOC parses the return type and function name from the function header immediately following the comment block.
<i>memberfunction</i>	AUTODOC parses the return type, class name, and function name from the function header immediately following the comment block.
<i>class</i>	Class name inserted into field 1.
<i>enum</i>	Enumeration name inserted into field 1.
<i>const</i>	Constant type and name inserted into fields 1 and 2.
<i>struct</i>	Structure tag name inserted into field 1.
<i>bsub</i>	"Sub" keyword plus any modifiers (Private, Public, etc.) inserted into field 1; subroutine name inserted into field 2.
<i>bfunc</i>	"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.

### Example

The following PARSESOURCE item defines function parsing:

```
.parsesource=function
```

### See Also

[\[TOPIC\]](#)  
[PARAGRAPH-IF](#)  
[CONTEXT](#)  
[ORDER](#)  
[PARSESOURCE](#)  
[TAG](#)

## 4.30

## [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

<i>sName</i>	Specifies the name of the tag. Type any name up to 63 characters, with no embedded spaces, tabs, commas, or semicolons.
<i>sOutputType</i>	Specifies the output type for the topic formatting block. Use one of the following strings:
<i>doc</i>	Specifies the formatting block is used for paper (Word document) output. The formatting block with this <i>sOutputType</i> value is used if the user specifies the /RD command-line flag.
<i>help</i>	Specifies the formatting block is used for help (WinHelp/Viewer topic file) output. The formatting block with this <i>sOutputType</i> value is used if the user specifies the /RH command-line flag.
<i>both</i>	Specifies the formatting block is used for both document and help output.
<i>nFields</i>	Specifies the number of fields in the tag. Type a number from 1 to 6.
<i>nSortLevel</i>	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.
<i>sTopicNameComponents</i>	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 <i>sTopicNameComponents</i> 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](#)



## 5 AUTODOC Tags

### 5.1 The @doc Tag

#### @doc

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

The **@doc** tag notifies the AUTODOC parser of the presence of AUTODOC tag blocks within a source file. The **@doc** tag also defines AUTODOC identifiers used to determine which topics to extract from the source file. The identifiers established by a **@doc** tag remain in force for all AUTODOC 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 AUTODOC 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](#)".

### 5.2 Topic Tags

Topic tags identify the beginning of an AUTODOC 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 AUTODOC topic tags:

#### 5.2.1 C Topics

These topic tags are used with C elements:

Table 5-1. C topic tags

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

## 5.2.2

### C++ Topics

These topic tags are used with C++ elements:

Table 5-2. C++ Topics

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

## 5.2.3

### OLE2 Topics

These topic tags are used with OLE2 elements:

Table 5-3. OLE2 Topics

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

## 5.2.4

### BASIC Topics

These topic tags are used with Visual Basic elements:

Table 5-4. BASIC Tags

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

## 5.2.5

### Table of Contents and Overview Topics

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

Table 5-5. Table of Contents and Overview Topics

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



## 5.3 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 AUTODOC paragraph tags:

### 5.3.1 C Tags

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

Table 5-6. C Tags

Tag	Usage
@emem	Enumeration members
@field	Structure fields
@rc	return code description
@parm	Parameters
@parmvar	Variable-length parameter list
@output	Introduces return code section
@globalv	Global variables (used in @module topic)

### 5.3.2 C++ Tags

These paragraph tags are used within C++ topics:

Table 5-7. C++ Tags

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

### 5.3.3 OLE2 Tags

These paragraph tags are used within OLE2 topics:

Table 5-8. OLE2 Tags

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

Table 5-8. OLE2 Tags

Tag	Usage
@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.

### 5.3.4

## BASIC Paragraphs

These paragraph tags are used with Visual Basic elements:

Table 5-9. BASIC paragraphs

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

## 5.4

## Comments and Annotations

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

Table 5-10. Comments and Annotations

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

### 5.4.1

## Miscellaneous

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

Table 5-11. Miscellaneous Tags

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

Table 5-11. Miscellaneous Tags

Tag	Usage
@head2	Heading level 2
@head3	Heading level 3
@end	Ends AUTODOC parsing within the comment block

## 5.4.2 Additional O/S kernel Tags

Table 5-12. Additional O/S kernel tags

Tag	Usage
@bug	describes a bug fix within the code
@fix	describes an entry of the version history
@version	describes the module version (in module)
@step	describes a step documentation of a module
@code	switches to <BXmp> paragraph style to enter pseudo code. A marker is generated to identify the pseudo code
@cont	continues the preceeding tag, although some source statements are present between the last statement (typically a @step tag) and the present statement

## 5.4.3 Special Sequence File Tags

Table 5-13. O/S kernel Sequence File Tags

Tag	Usage
@testcase	describes a test case name
@scall	describes a called sequence
@description	describes a test case
@result	describes the expected result of a test case

## 5.5 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:

### 5.5.1

## C Tags

These tags are used for C constructs:

Table 5-14. 'C' Tags

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

### 5.5.2

## C++ Tags

These tags are used for C++ constructs:

Table 5-15. C++ Tags

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

### 5.5.3

## OLE2 Tags

These tags are used for OLE2 constructs:

Table 5-16. OLE2 Tags

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

### 5.5.4

## Graphics

This tag lets you insert a bitmap file, in O/S kernel use the <fig and <tbl tag instead. Refer to [6.4 "Figures and Tables" on page 6-142](#).

Table 5-17. Graphis Tags

Tag	Usage
<bmp	Bitmap graphic file.
<fig	Pre-designed figure, written in FrameMaker reference file
<tbl	Pre-designed table, written in FrameMaker reference file

## 5.5.5

## Special Characters

These tags represent special characters:

Table 5-18. Special characters text tags

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
<or	Or-character, instead of field delimiter

## 5.6

## Tags in alphabetical order

### 5.6.1

### @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](#)

## 5.6.2 @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](#)

## 5.6.3 @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](#)

## 5.6.4 @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

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

## See Also

[@bparm](#), [@bsub](#)

## 5.6.5 @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](#)

## 5.6.6 @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

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

## See Also

[@bfunc](#), [@bparm](#)

## 5.6.7

### @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

AUTODOC 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](#)

## 5.6.8 @catch (topic-level)

The [@catch](#) tag is a topic tag used to document the error handler of a function.

## Syntax

[@catch](#) *description*

## Example

The following example shows the [@catch](#) tag:

```
// @catch The return code is propagated to the caller
```

which appears as

## 12.9.42 Error Handler

The return code is propagated to the caller

## Comments

**This step was added for particular use in O/S kernel.**

## 5.6.9 @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](#)

### 5.6.10 @cmember (paragraph-level)

The [@cmember](#) tag is used within the [@class](#) tag to provide a simple description of class members. 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, AUTODOC 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](#)

### 5.6.11 @comm (paragraph-level)

The `@comm` tag is used to add comments to any AUTODOC 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`

### 5.6.12 @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;
```

### 5.6.13 @code (topic level)

The `@code` tag is a topic tag used to write pseudo code. A marker is generated in order to identify the pseudo code and to allow to refer to it from other parts of the document.

#### Syntax

`@code` text

#### Example

The following example shows the `@code` tag:

```
// @code
// VerifyAccessClass();
// TestParameter(input);
```

#### Comments

To continue with normal text use the `@cont` tag for further description

**This step was added for particular use in O/S kernel.**

#### See also

`@excont (topic-level)`

### 5.6.14 @cont (topic-level)

The `@cont` tag is a topic tag used to resume a previously written topic, typically a `@step` topic.

## Syntax

**@cont** text

## Example

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

```
// @step 4 | normalize the value | The value in the register is
// normalized in order to comply with the hardware specification

    N = normalize(RegVal;

// @cont The normalized value N will be inverted to support the
// computation
```

which appears as

### Step 4 : Normalize the value

The value in the register is normalized in order to comply with the hardware specification.

The normalized value N will be inverted to support the computation.

## Comments

**This step was added for particular use in O/S kernel.**

## See also

[“@code \(topic level\)”](#)

## 5.6.15

### @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>
```

## 5.6.16

### @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 AUTODOC, 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](#)

## 5.6.17

### @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 AUTODOC, for an example.

#### Syntax

[@contents2](#) | Contents Heading | Contents Paragraph

#### See Also

[@subindex](#) [@contents1](#)

## 5.6.18

### @description (topic-level)

The [@description](#) tag describes the functionality of a test case

#### Syntax

[@description](#) <description text>

#### Example

The following example shows the [@description](#) tag

```
*-----
* @testcase FileCreate
*
* @description Testing the FileCreate command ...
*-----
```

#### See Also

[@testcase](#) (paragraph-level)

## 5.6.19

### @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 AUTODOC builds.

## See Also

[@todo](#)

## 5.6.20

### @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](#)

## 5.6.21

### @end (paragraph-level)

Empty tag used to terminate the AUTODOC section of a comment. Insert the [@end](#) tag at the end of the AUTODOC tags, and any text following the tag will be ignored.

## Syntax

[@end](#)

## 5.6.22

### @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](#)



## 5.6.23

### @ex (paragraph-level)

The **@ex** tag is used to document example source code. Use the similar **@tex** 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.

**Note:** Use the **@ex** tags instead of the `<para BXmp>` style selection in order to inform the AUTODOC program to keep leading and trailing blanks in the following text. A similar text tag **<ex[1..4]>** is available too for the same purpose

The difference between the **@ex** tag and the **<ex[1..4]>** text tag is, that the **@ex** tag will generate a special section in the output to describe an example (like it is shown in this document). However the **<ex[1..4]>** text tag will allow you to generate an example in any text flow.

**Hint:** The **@ex** tag will use the `BXmp` paragraph style. If you like to indent an example you may write **@ex<ex2>** *Example text*

### Syntax

**@ex** description | example

### Comments

Text in the example field can include special AUTODOC 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 ) );
```

## 5.6.24

### @excont (topic-level)

The **@excont** tag is a topic tag used to resume a previously written example topic. In particular the text after the **@excont** tag is immediately appended to the previous example text. It will also have the `BXmp` font.

### Syntax

**@excont** text

### Example

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

```
; @step 4 | Normalize the value | The value in the register is
; normalized and inverted. The following example show the command flow
;
; @code
; N = normalize(RegVal);

    lda #RegVal
    call Normalize

; @excont
```

```

;   N *= -1;
    call   Invert

; @cont Then the program returns

```

which appears as

## Step 4 : Normalize the value

The value in the register is normalized and inverted. The following example show the command flow

```

N = Normalize(RegVal)
N *= -1;

```

Then the program returns

## Comments

**This step was added for particular use in O/S kernel.**

## See also

["@code \(topic level\)"](#)

## 5.6.25

### @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](#)

## 5.6.26

### @rc (paragraph-level)

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

#### Syntax

**@rc** name | description

#### Example

The following example shows the **@rc** tag (this time used with the **@output** tag):

```
// @func This function compares two strings.
//
// @output Returns one of the following values:
//
// @rc -1 | If <p szStr1> is smaller.
// @rc 1 | If <p szStr2> is smaller.
// @rc 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](#)

## 5.6.27

### @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 AUTODOC 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.
//
// @output Returns one of the following values:
//
// @rc -1 | If <p szStr1> is smaller.
// @rc 1 | If <p szStr2> is smaller.
// @rc 0 | If <p szStr1> and <p szStr2> are the same.

int strcmp(char *szStr1, char *szStr2)

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

`@output @parm @comm @ex @xref @flag`

## See Also

`@cb`

## 5.6.28

### `@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
 *
 * AUTODOC 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;
```

## 5.6.29

### `@group` (paragraph-level)

The `@group` tag is used to add a subheading within any AUTODOC 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**

## 5.6.30

### **@head1 (paragraph-level)**

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

## Syntax

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

## 5.6.31

### **@head2 (paragraph-level)**

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

## Syntax

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

## 5.6.32

### **@head3 (paragraph-level)**

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

## Syntax

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

## 5.6.33

### **@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 AUTODOC 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 ) );
```

## 5.6.34

### **@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](#)

### 5.6.35 @index (paragraph-level)

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

**Note:** As FrameMaker generates its own index, this tag might not be used in the O/S kernel documentation.

## 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](#)

### 5.6.36 @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](#)

### 5.6.37 @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](#)

### 5.6.38 @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`

## 5.6.39

### `@menu` (topic-level)

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

## Syntax

`@menu` class\_name | enumeration\_name | description

## Example

The following examples show the `@menu` 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,menu 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"

## 5.6.40

### `@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`

## 5.6.41

### **@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`

## 5.6.42

### **@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
*
* AUTODOC 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](#)

## 5.6.43

### **@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](#)

## 5.6.44

### **@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"

### 5.6.45

#### @normal (paragraph-level)

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

## Syntax

[@normal](#) Paragraph text...

### 5.6.46

#### @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.
//
// @output Returns one of the following values:
//
// @rc -1 | If <p szStr1> is smaller.
```

```

// @rc 1 | If <p szStr2> is smaller.
// @rc 0 | If <p szStr1> and <p szStr2> are the same.

int strcmp(char *szStr1, char *szStr2)

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

[@flag](#) [@func](#) [@mfunc](#) [@method](#)

## 5.6.47

### @pre (paragraph-level)

The [@pre](#) tag is used to document preconditions

## Syntax

[@pre](#) description

## Comments

The [@pre](#) tag typically is at the end of a function header declaration to point out whether there are special conditions to be considered on entry.

## 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.
//
// @pre szStr must not be zero
//
// @output Returns one of the following values:
//
// @rc -1 | If <p szStr1> is smaller.
// @rc 1 | If <p szStr2> is smaller.
// @rc 0 | If <p szStr1> and <p szStr2> are the same.

int strcmp(char *szStr1, char *szStr2)

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

[@flag](#) [@func](#) [@mfunc](#) [@method](#)

## 5.6.48

### **@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](#)

## 5.6.49

### **@result (topic-level)**

The [@result](#) tag identifies a result description a sequence file.

## Syntax

[@result](#) <description>

## Example

The following example shows the [@result](#) tag

```
*-----
* @testcase FileCreate
*
* @description Testing the FileCreate command ...
*
* @result We expected rc=9000 on successful completion
*         If there is no memory found the return code
*         is <rc>rc = 6985<ef>
*-----
```

## See Also

[@testcase](#) (paragraph-level), [@sequence](#) (paragraph-level), [@description](#) (topic-level)

## 5.6.50

### **@output (paragraph-level)**

The [@output](#) tag is used to document return values of functions and messages.

## Syntax

[@output](#) 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 **@output** tag:

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

### 5.6.51 @rc (paragraph-level)

The **@rc** tag is used to document the HRESULT status codes and their meanings.

#### Syntax

**@rc** status code | description

#### Example

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

```
// @rc S_OK | The operation succeeded.
```

### 5.6.52 @scall

The **@scall** tag identifies a sequence call in a sequence file.

#### Syntax

**@scall** <testcase name>

#### Example

The following example shows the **@scall** tag

```
*-----
* @testcase FileCreate
*
* @scall InitFiles | is called in order to initialize all variables
*-----
```

#### See Also

@testcase (paragraph-level), @description (topic-level)

### 5.6.53 @sequence (paragraph-level)

The **@sequence** tag describes a sequence file

#### Syntax

**@sequence** <sequence name>

#### Example

The following example shows the **@sequence** tag

```
*-----
* @doc
* @sequence FileCreate.SEQ
*-----
```

## See Also

@description (topic-level), @testcase (paragraph-level)

## 5.6.54

### @step (topic-level)

The **@step** tag is a topic tag used to document a processing step in the low level design of a command flow.

## Syntax

**@step** number | Header | description

## Example

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

```
// @step 4 | normalize the value | The value in the register is
// normalized in order to comply with the hardware specification

N = normalize(RegVal;

// @cont The normalized value N will be inverted to support the
// computation
```

which appears as

### Step 4 : Normalize the value

The value in the register is normalized in order to comply with the hardware specification.

The normalized value N will be inverted to support the computation.

## Comments

**This step was added for particular use in O/S kernel.**

## See also

“@code (topic level)”

## 5.6.55

### @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](#)

## 5.6.56

### @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](#)

## 5.6.57

### @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.
```

## 5.6.58

### @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](#)

## 5.6.59

### **@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](#)

## 5.6.60

### **@testcase (paragraph-level)**

The **@testcase** tag identifies a test case in a sequence file.

## Syntax

**@testcase** <testcase name>



## Example

The following example shows the `@testcase` tag

```
*-----  
* @testcase FileCreate  
*  
* @description Testing the FileCreate command ...  
*-----
```

## See Also

`@description` (topic-level)

## 5.6.61

### `@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`

## 5.6.62

### `@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](#)

### 5.6.63

#### **@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](#)

### 5.6.64

#### **@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 \(topic-level\)](#)

### 5.6.65

#### **@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>
```

## 5.7 Text Tags

Text tags may appear in any text of a paragraph of a body. They are a powerful method to change the font, paragraph style, insert special characters or figures and tables into the output text.

### 5.7.1 **<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.

In O/S kernel documentation this tag is not used. Use the **<fig** tag instead.

### 5.7.2 **<c**

The **<c** tag is used to identify references to classes.

#### Syntax

**<c** class name>

### 5.7.3 **<cp**

The **<cp** tag is used to generate a copyright symbol (©).

#### Syntax

**<cp>**

### 5.7.4 **<ct**

The **<ct** tag is used to identify constants

#### Syntax

**<ct** *constant name*>

#### Comment

**This is a new tag added for use with FrameMaker (HS)**

### 5.7.5 **<date**

Inserts the date of the AUTODOC build.

#### Syntax

**<date>**

### 5.7.6 **<e**

The **<e** tag is used to identify references to structure members.

#### Syntax

**<e** type name.member name>

### 5.7.7

#### <ef

The **<ef** tag is used to end a font and return to the default font of a paragraph.

#### Syntax

This text is normal but**<f Blue>**this text is blue**<ef>** and this is normal again

#### Comment

**This is a new tag added for use with FrameMaker (HS)**

### 5.7.8

#### <em-

The **<em-** tag is used to generate an em dash character (-).

#### Syntax

**<em->**

#### See Also

**<en-**

### 5.7.9

#### <en-

The **<en-** tag is used to generate an en dash character (-).

#### Syntax

**<en->**

#### See Also

**<em-**

### 5.7.10

#### <ex

The **<ex>** tag is used in order to switch to the *BXmp* font within the current paragraph style. Leading and trailing blanks in a line will not be removed except the first blank following the comment character (// or ;).

#### Syntax

**<ex>**Example Text  
Next Example Line  
**<ef>** : to switch back to the default paragraph style, you can also use any  
**<para ...>** tag to change back to normal text mode.

**Note:** Use the **<ex>** text tag instead of the **<f BXmp>** tag. The **<ex>** tag will avoid the removal of leading and trailing spaces except the first blank following the comment character (// or ;) - in order to have a correctly formatted layout of the example.

### 5.7.11

#### <ex[1..4]>

The **<exn>** tag is used in order to switch to the *BXmpn* paragraph style. Leading and trailing blanks in a line will not be removed except the first blank following the comment character (// or ;).

#### Syntax

**<ex1>**Example Text  
Next Example Line  
**<iex>** : to switch back to Body paragraph style, you can also use any **<para ...>** tag to change back to normal text mode.

### 5.7.12

#### <f

The **<f** tag is used to start a font within the current text flow

#### Syntax

This text is normal but**<f Blue>**this text is blue**<ef>** and this is normal again

## Comment

**This is a new tag added for use with FrameMaker (HS)**

The former version of the <f font (reference to functions and macros) was deleted

## 5.7.13

### <fc

The <fc tag is used to abbreviate the <f Pv> statement.

## Syntax

<fc>*any text, typically a function() name*<ef>

## Description

This is a simple shortcut of <f Pv>*any text*<ef>. As it appears frequently in text, this tag was created to increase the readability of the source.

## Comment

**This is a new tag added for use with FrameMaker (HS)**

## 5.7.14

### <fig

The <fig tag is used to insert a figure into the output file

## Syntax

At this point <fig *TestFigure01*> an anchor of a reference figure should be added

## Description

The figure that is referred by *TestFigure01* has to be created in a FrameMaker document which was exported and saved as .MIF file. This .MIF file shall be specified in with the /b option during the AUTODOC invocation. Then the figure (with the caption <Figure *TestFigure01* will be stripped from the reference file and inserted into the output document. Refer to 6.4 “Figures and Tables” on page 6-142 for a more detailed explanation.

## Comment

**This is a new tag added for use with FrameMaker (HS)**

## 5.7.15

### <filename

Inserts the source filename.

## Syntax

<filename>

## 5.7.16

### <filepath

Inserts the source file path.

## Syntax

<filepath>

## 5.7.17

### <ge

The <ge tag is used to generate a greater-equal symbol (≥).

## Syntax

<ge>

## See Also

<gt, <lt, <le

## Comment

**This is a new tag added for use with FrameMaker (HS)**

### 5.7.18

#### <gt

The **<gt** tag is used to generate a greater-than symbol (>).

#### Syntax

**<gt>**

#### See Also

<lt, <ge, <le

### 5.7.19

#### <iex

The **<iex** tag is used to end an example (started with **<ex[1..4]>**). This tag will set the font back to 'BodyAfterHead' to allow the text to continue. However it ends the paragraph with a HardReturn in order to have a correct layout of the last line of the example.

Use this tag to end an example, when text follows the example within the same paragraph.

#### Syntax

**<iex>**

#### See Also

<ex, <nex

### 5.7.20

#### <im

The **<im** tag is used to identify references to interface methods.

#### Syntax

**<im** interface method>

### 5.7.21

#### <kw

The **<kw** tag is used to highlight 'C' keywords

#### Syntax

**<kw>**if**<ef>** (test > 0) ....

#### Comment

**This is a new tag added for use with FrameMaker (HS)**

### 5.7.22

#### <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**

### 5.7.23

#### <le

The **<le** tag is used to generate a less-equal symbol (≤).

#### Syntax

**<le>**

## See Also

<gt, <lt, <ge

## Comment

**This is a new tag added for use with FrameMaker (HS)**

## 5.7.24

### <lq

Inserts a left quote.

## Syntax

<lq>

## See Also

<rc

## 5.7.25

### <lt

The <lt tag is used to generate a less-than symbol (<).

## Syntax

<lt>

## See Also

<gt, <ge, <le

## 5.7.26

### <m

The <m tag is used to identify references to messages.

## Syntax

<m message name>

## 5.7.27

### <md

The <md tag is used to identify references to class member data.

## Syntax

<md class name::member name>

## 5.7.28

### <mf

The <mf tag is used to identify references to class member functions.

## Syntax

<mf class name::member function name>

## 5.7.29

### <notequal>

The <notequal tag is used to generate a less-equal symbol (≠).

## Syntax

<notequal>

## See Also

<le, <lt, <gt, <ge

## Comment

**This is a new tag added for use with FrameMaker (HS)**

### 5.7.30

#### <nex

The **<nex>** tag is used to end an example (started with **<ex[1..4]>** or **<ex>**). This tag does not switch the font back to 'BodyAfterHead', as it assumes that the next @tag will provide its own paragraph style definition. However it ends the paragraph with a HardReturn in order to have a correct layout of the last line of the example.

Use the **<nex>** tag, if no other text follows the paragraph

#### Syntax

**<nex>**

#### See Also

**<ex>**, **<iex>**

### 5.7.31

#### <nl

The **<nl>** tag is used to generate a new line character.

#### Syntax

**<nl>**

### 5.7.32

#### <or

The **<or>** tag is used to insert a 'I' - OR character.

#### Syntax

**<or>**

#### Description

This tag is required to insert the 'OR' character I into the documentation. Typically this character is used as field delimiter, hence it requires an appropriate text tag to be inserted.

#### Comment

**This is a new tag added for use with FrameMaker (HS)**

### 5.7.33

#### <p

The **<p>** tag is used to identify references to parameters.

#### Syntax

**<p>** parameter name>

### 5.7.34

#### <rc

The **<rc>** tag is used to abbreviate the **<f BPk>** statement.

#### Syntax

**<rc>**9000**<ef>**

#### Description

This is a simple shortcut of **<f BPk>***any text***<ef>**. It is used to express any return values in O/S kernel documentation. As it appears frequently in text, this tag was created to increase the readability of the source.

#### Comment

**This is a new tag added for use with FrameMaker (HS)**

### 5.7.35

#### <rq

Inserts a right quote.



## Syntax

`<rq>`

## See Also

`<le`

## 5.7.36

### **<rtm**

The `<rtm` tag is used to generate a registered trademark symbol (®).

## Syntax

`<rtm>`

## See Also

`<tm`

## 5.7.37

### **<sub**

The `<sub` tag is used to add a subscript

## Syntax

A`<sub i>` produces  $A_i$

## Comment

**This is a new tag added for use with FrameMaker (HS)**

## 5.7.38

### **<sup**

The `<sup` tag is used to add a superscript

## Syntax

e`<sup x-1>` produces  $e^{x-1}$

## Comment

**This is a new tag added for use with FrameMaker (HS)**

## 5.7.39

### **<sv**

The `<sv` tag is used to insert a system variable.

## Syntax

`<sv sysvar>`

## Comment

**This is a new tag added for use with FrameMaker (HS)**

## 5.7.40

### **<t**

The `<t` tag is used to identify references to structure and enumeration types.

## Syntax

`<t type name>`

## 5.7.41

### **<tab**

Inserts a tab symbol.

## Syntax

`<tab>`

## 5.7.42

### **<tbl**

The `<tbl` tag is used to insert a table into the output file

## Syntax

At this point `<fig TestTable01>` an anchor of a reference table should be added

## Description

The table that is referred by `TestTable01` has to be created in a FrameMaker document which was exported and saved as .MIF file. This .MIF file shall be specified in with the `/b` option during the AUTODOC invocation. Then the table (with the caption `<Table TestTable01` will be stripped from the reference file and inserted into the output document. Refer to [6.4 “Figures and Tables” on page 6-142](#) for a more detailed explanation.

## Comment

**This is a new tag added for use with FrameMaker (HS)**

## 5.7.43

### `<tm`

The `<tm` tag is used to generate a trademark symbol <sup>TM</sup>

## Syntax

`<tm>`

## See Also

`<rtm`

## 5.7.44

### `<v`

The `<v` tag is used to abbreviate the `<f BPv>` statement.

## Syntax

`<v>`*any text, typically a variable name*`<ef>`

## Description

This is a simple shortcut of `<f BPv>`*any text*`<ef>`. As it appears frequently in text, this tag was created to increase the readability of the source.

## Comment

**This is a new tag added for use with FrameMaker (HS)**

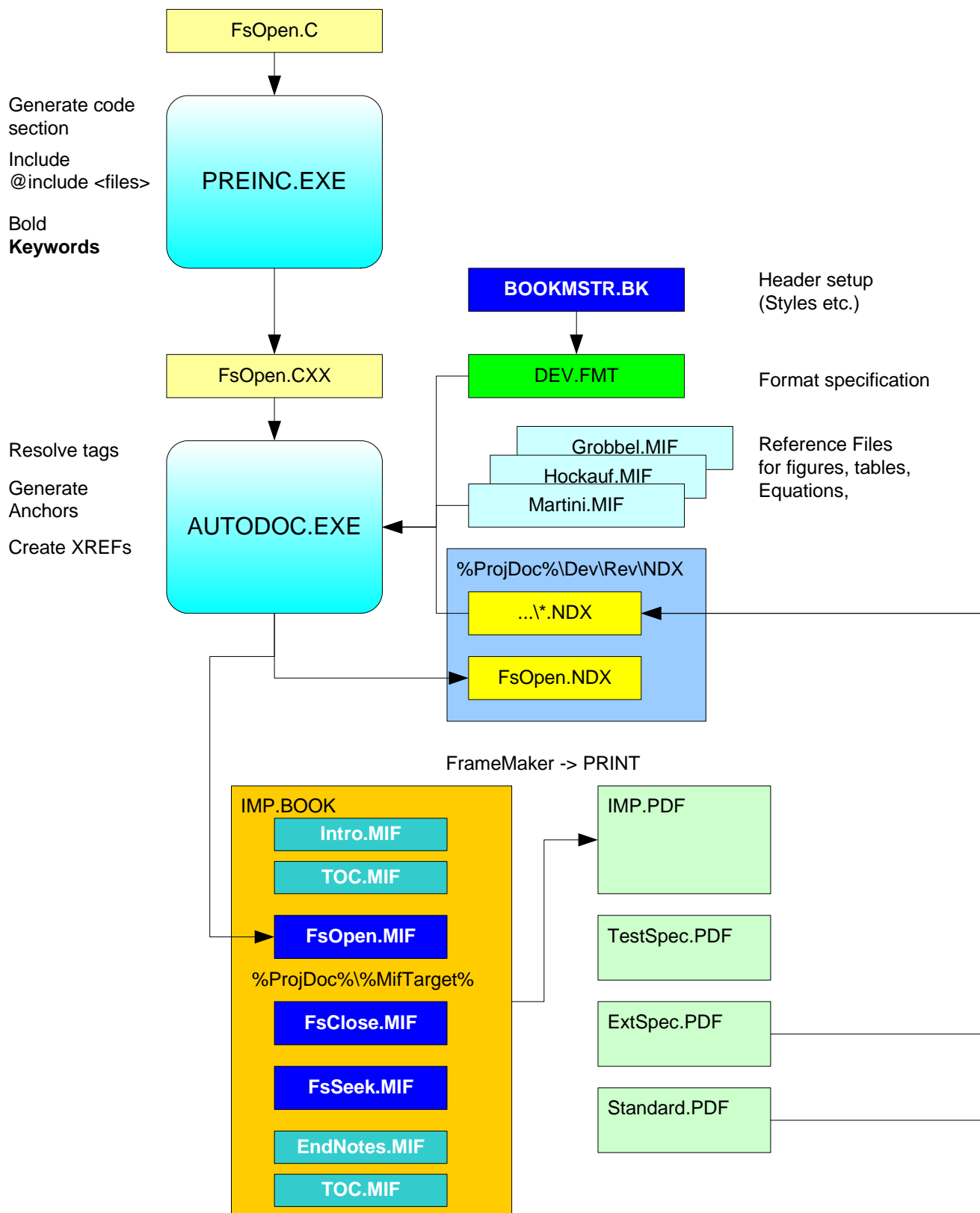


Figure 6-1. Autodocumentation process overview

## 6.1 Header Section

### @doc

Any document using the AUTODOC feature has to start with the **@doc** tag. Any other tags will not be interpreted unless the **@doc** tag was found in the document.

The **@doc** tag may be followed by a qualifier

**@doc** EXTERNAL will output the following definitions, if the EXTERNAL flag is defined at runtime of AD.EXE. @doc INTERNAL will output the subsequent definitions

### @module *name / description*

The **@module** tag describes the name and the function of the whole file. A module is an ASCII file containing one or more functions

The **@module** tag, when encountered in the source, lists several paragraphs in order to describe the version (**@version** tag) and change history (**@fix** tag) of the file.

### @version *version # / driver #*

The **@version** tag indicates the version.driver information of the module.

### @fix *data / version /author / description*

The **@fix** tag indicates the change history of the module. It shall be used in order to record changes to a file after its initial release

## 6.2 Code section

### 6.2.1 Paragraph tags

#### @func *type / name / desc*

The **@func** tag describes an assembler function in the layout of a typical 'C' function. It is typically followed by a list of parameters (**@parm**) and a description of the return values (**@rdesc / @flag**).

The *type* parameter describes the type of the output value of the function. This does not always apply to assembler functions. A possible solution to this dilemma could be using the MATHLAB notation of a vector output :

```
@func [int a, char* b] | TestFun | This function demonstrates multiple variable output
```

The generated type will follow this definition.

#### @parm *type / name / desc*

The **@parm** tag may appear multiple times, each time describing an input parameter of the function.

#### @rdesc *desc*

The **@rdesc** topic preceeds a list of **@flag** tag that describe a particular set of values to be returned.

#### @flag *value / description*

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

### @vuln title | description

The @vuln is used to indicate a possible vulnerability in the code. Frequently programmers can identify potential vulnerabilities (timing attack) already on creating the code. The depth of leakage, however can only be investigated later. The @vuln tag can serve as warning to observe these parts in the code.

### @bug test reference | description

The @bug tag allows the description of bugs and fixes in the code. A list of bugs is collected for each function in order to receive an overview for the changes done to the code. The 'test reference' parameter allows to indicate a reference to the appropriate test scenario.

### @step Number | Header | description

The @step tag is used to comment a processing step. It is output to MIF file as Head3 paragraph. The 'description' field is output as 'BodyAfterHead' paragraph.

### @cont description

The @cont tag is used in order to continue the text of a preceding tag. It is formatted as 'Body' paragraph by default, however the paragraph format may be changed using the <para [style] > text tag.

The @cont text is often used to continue the description of a step in the code. After several statements you might want to describe what's going in the code. Using the @cont tag allows you to add the next comment and make it appear right after your previous comment given in a preceding @step. It will also continue other tags than @step only.

This tag is helpful to write your comment text located at the appropriate code statement. In the documentation the text keeps together with the text of the preceding (e.g. @step) tag.

### @catch description

The @catch tag is used in order to describe an error handler. The text in *description* describes the function of the catch section of a module.

## 6.2.2 Text tags

Text tags are used in order to change the formatting style within paragraphs

<para [stylename] >

sets the paragraph style. <para BodyAfterHead> makes the paragraph appear in this style.

**Note:** This tag is case sensitive and will not work if not written correctly.

<f [fontname]>

sets the current font. <f BPv> sets the 'BPv' font, <f Blue> set the color blue until the <ef> tag or another paragraph is found.

**Note:** Do not set <fBXmp...> fonts. Use the following <ex...> tags instead.  
Reason: The <ex ..> tags are more powerful and control the parser to take in the formatting of your file exactly without removing heading or trailing blanks. Only the first blank after the comment character (e.g. // or ; ) will be ignored.

<ef> Ends the font and returns to the normal font.

<ex> sets the BXmp font and avoids the removal of blanks in the beginning of a line. The paragraph style will be kept

<ex1> sets the BXmp paragraph and avoids the removal of blanks in the beginning of a line

<b>&lt;ex2&gt;</b>	sets the BXmp2 paragraph and avoids the removal of blanks in the beginning of a line
<b>&lt;ex3&gt;</b>	sets the BXmp3 paragraph and avoids the removal of blanks in the beginning of a line
<b>&lt;ex4&gt;</b>	sets the BXmp4 paragraph and avoids the removal of blanks in the beginning of a line
<b>&lt;fig name&gt;</b>	insert a figure anchor. See next paragraph
<b>&lt;rc&gt;</b>	sets the BPk font in order to describe a return code. Turned off with the <b>&lt;ef&gt;</b> tag.
<b>&lt;sv name&gt;</b>	insert a system variable
<b>&lt;tbl name&gt;</b>	insert a table anchor. See next paragraph
<b>&lt;xref name&gt;</b>	insert a cross reference anchor.
<b>&lt;v&gt;</b>	sets the BPv font in order to describe a variable. Turned off with the <b>&lt;ef&gt;</b> tag.

## 6.3 Special characters and symbols

Special characters may be inserted in two ways. In order to create any special ASCII code you may use the **^ prefix** to supply the two letter ASCII code.

This is in particular important using symbols. E.g. the greek letter  $\Phi$  can be generated by  
 $\Phi$  = <f Symbol>F<ef>

or with the prefix ^ using

$\Phi$  = <f Symbol>^46 <ef>

**Note:** It is essentially important to add a space character after the two hex nibbles. So the '46' in the example above is followed by a space before the **<ef>** tag ends the font.

The first option, is recommended, if the character has an ASCII value less than 128 ('80'x).

The second option is more appropriate, if the ASCII value of the character to be generated has an ASCII value greater than 127.

The ASCII values to directly correspond with the following table, taken from David Toll's famous BOOKMASTER guide which is added to this document as [Appendix 9. "FrameMaker Character Sets" in volume 2 page 9-1..](#)

## 6.4 Figures and Tables

Another new feature has been added to the original AUTODOC program in order to support figures and tables in FrameMaker.

It is considered difficult to create a sophisticated figure or table layouts in ASCII text. Therefore another approach was taken.

**Figures and tables are created in a separate reference file <myname>.FM.**

### 6.4.1

## Step 1: Create the figures and tables in the reference file

You can link figures by reference in the `<myname>.fm` document. It is not necessary to imbed them. This has the advantage, that you can update your drawings without having to do the inclusion of the figure again.

In the reference file `<myname>.FM` the description part of a figure contains a special text identifier

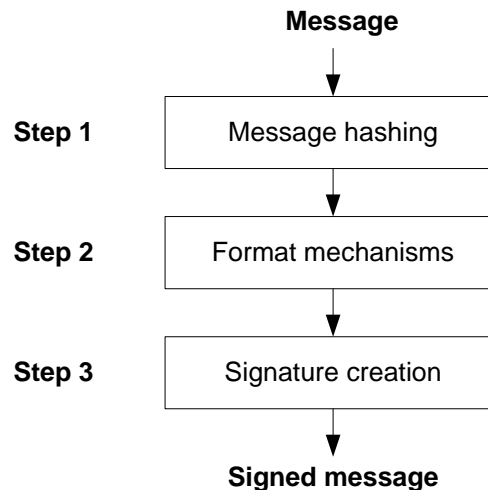


Figure 6-2. **<Figure MsgSign>** Signing a message with a SmarCard

The text identifier is formed in a **<Figure text>** construct.

A table is built in a similar way

Table 6-1. **<Table SignAct>** Signing Actions

Step	Action
Step 1	Message hashing
Step 2	Format Mechanisms
Step 3	Signature creation

Figures and tables such can be created entirely sophisticated and according to any choice of the editor.

### 6.4.2

## Step 2: Export the reference file to MIF

Use the 'SaveAs' function of FrameMaker in order to export the `<myfile>.fm` as `<myfile>.>.MIF`. This avoids any tampering of the AUTODOC with the original `<myfile>.FM` and creates an ASCII readable (MIF) version of your figures.

In particular linked (by reference) figures by reference in the `.FM` document, will be exported as OLE objects in the MIF file. Such the MIF file contains the full drawing, even if the `.FM` file used the 'link by reference' option.

### 6.4.3

## Step 3: Write the program source(s)

Use the `<fig name>` and `<tbl name>` text tags in order to insert any of your pre-designed figures into the final output document. name specifies the name that you gave a figure (table) in the caption.

Writing a text in the program (ASCII) file and using the `<fig name>/<tbl name>` text tag places the figure/table anchor exactly where the `<fig name>` or `<tbl name>` text tag was found.

The *name* variable specifies the same name as used in the FrameMaker <myfile>.FM caption of the appropriate figure.

The <fig *name*> in the program (ASCII) file corresponds to the <Figure *name*> entry in the FrameMaker <myfile>.FM file.

The <tbl *name*> in the program (ASCII) file corresponds to the <Table *name*> entry in the FrameMaker <myfile>.FM file.

The <xref *name*> in the program (ASCII) file corresponds to the <Xref *name*> entry in the FrameMaker <myfile>.FM file.

#### 6.4.4

### Step 4: Run AUTODOC with the /b option

When invoking AUTODOC use the option flag /b in order to inform the AUTODOC program to use your figure reference file <myfile>.MIF.

A typical AUTODOC invocation could be ...

```
F:\ASM>AD /f "f:\fmt\fm.fmt" /b "..\REF\<myfile>" test.asm
```

The extension .MIF will automatically be added to the <myfile> if no extension is given explicitly.

AUTODOC merges the figures and tables, equations and cross references into the output file wherever you refer them.

**Note:** A figure or table shall be reference only once in one or a set of files processed with AUTODOC. If the same figure is referenced twice, FrameMaker will complain for double usage of a figure, when the .MIF output file is imported.

If you want to use the same figure twice, you must copy it in the reference document <myfile> and supply with a different <Figure ....> text tag. Hence you may refer the copy within your source. This effect is no limitation as in writing a FrameMaker document you wouldn't be able to use one figure in two places either.

## 6.5

### Equations

The generation of an equation can be made with

$$\text{<eq TestEq1>Quota} = \text{sizeof(tMemBlock)} + \text{HdrSize} + \text{sizeof(IFINFO)} \quad (6-1)$$

in the reference document.

The source code will relate to this equation with the <eq TestEq1> text tag similar to the description of figures and tables above.

## 6.6

### Updating figures, tables and equations

In many cases, flow charts and figures in the O/S kernel OS are created with MicroSoft Visio 2000. The files <drawing>.VSD may be included as objects by reference into the FrameMaker Source. (File-Import-Object-Create From File-Link-Browse <\*.vsd>, do not forget to select the anchored frame previously created).

Changing such a <drawing>.VSD file will automatically update the FrameMaker .FM file, when opened next.

Whatever the way of creation is, after the reference file was brought up-to-date, in order to update the final output, you simply need to export the reference file again as <myfile>.MIF as discussed in 6.4.2 on page 6-143. Then run the AUTODOC program to create the final output source.



These few steps are considered affordable to support figures and tables without having too many tasks for the programmer.

Tables and Equations may be updated in the reference document directly.

## 6.7 Cross references

### 6.7.1 Cross references ex-source

If you want to generate a cross reference in your source to any external document, you need to use the `<xref name>` tag.

The reference document contains the actual reference, which is copied into the output document, when processing AUTODOC.

If the reference document contains

`<xref AsmExample>`6.10 “Assembler Code Example” on page 6-146

then the source code shall contain a statement

; You may also refer to `<xref AsmExample>`

During processing, AUTODOC will find the cross reference and insert it into the output document. Hence the output document will contain outbound cross references with a freshness of the reference document.

### 6.7.2 Cross references in-source

Cross references may be made to

- Headings
- Figures
- Tables
- Equations

AUTODOC prepares these paragraph styles with unique markers. The unique identifier of these markers is derived by a hash function from the heading text. Hence these markers will be exactly the same as long as the heading text (like a structure name etc.) is not changed. Also the heading of a `@step` statement is, for instance, part of this hash.

In order to generate a reference into an AUTODOC generated file, you simply have to make a cross reference into the appropriate .MIF or .FM file, depending what final file type you choose for your documentation

**Note:** We recommend the conversion of a generated .MIF file to .FM (File-SaveAs “.FM”). for backup security reasons. This way it is impossible by an accidental AUTODOC invocation to delete the actual documentation.

Typically FrameMaker alters the target document of reference, because it has to add a marker to the file. As AUTODOC has pre-generated all markers on headers, figures and tables, on establishing a cross reference to an AUTODOC generated file, the target file does not need to be changed. FrameMaker recognizes the existence of the marker and indeed does not alter the target file.

This, however has the effect, that ‘inbound’ cross references are constant and reliable unless the header title / tag fields are changed in the target document.

If this is the case. the cross reference for the altered tags would have to be updated. Therefore it is recommended to do the in-source cross references in a later stage of the source, where the tag descriptors are trustable.

## 6.8 System variables

Use the `<sv varname >` text tag to insert any system variable at any place in your text.

## 6.9 Adding system variables

You may list your personal additional system variables in the reference file. The global system variables defined in BOOKMSTR.MIF will be present anyway. So you would only have to enter your additional variables into the reference file. You may use the DT/DD tags like it is proposed here, however you should not use special fonts within the text, because this generates 'noise' while these variables are extracted from the reference file (in its MIF representation).

The additional system variables will be available in any output document.

```
<sv.Name>psmgr
      <sv.Def>persistent storage manager
```

## 6.10 Assembler Code Example

The following example demonstrates the typical tags used in Assembler files.

```
;*****
;
; Caernarvon Secure Operating System
;
; IBM Research Division, T. J. Watson Research Center
;
; (C) COPYRIGHT IBM CORP 2000, 2001, 2002
;
; LICENSED MATERIALS - PROPERTY OF IBM
;
; All Rights Reserved
;
; U.S. Government Users Restricted Rights - Use, duplication or
; disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
;
; See Copyright Instructions (Form G120-2083)
;
; Origins:          IBM T. J. Watson Research Center
;
; MODULE TYPE:      ASM Source File
;
; RESTRICTIONS:      DPA tests not yet finished
;
; CREATION DATE:     Created Nov 2000 by H. Scherzer, IBM Watson Research
;                    Center, Hawthorne
;
;*****
; @doc
; @module FASTDIV.AXA | implements the FAMEX FastDiv function
; @version 3 | 2
;
; @fix 01-Nov-00 | 1.0 | HS | Module created
; @fix 13-Dec-00 | 2.1 | HS | TempSpace wasn't cleared correctly (P_DIV_01_05)
; @fix 11-Jun-01 | 3.0 | ml | Convert to RIDE.
; @fix 06-Aug-01 | 3.0 | je | Convert from segment 15 offsets to physical addresses
; @fix 22-Aug-01 | 3.0 | DCT | Clean up include files
; @fix 18-Sep-01 | 3.1 | HS | Fixed save/restore of tempvars beyond 1KB
```

```

; @fix 04-Oct-01 | 3.1 | je | Add undecorated entry point
; @fix 30-Nov-01 | 3.2 | DCT | Change ROM_CODE to CRYPTO_CODE
;
; @fix 12-Mar-02 | 4.0 | HS | migration to Caernarvon kernel, AD tags added
;                               the $-tags for the scanhdr.exe function have been
;                               changed to ampersand tags in order to comply with
;                               the AUTODOC program
;
;***** End of Header *****
;$CASE
;$ZPAGE

$include (crypto.inc)
$include (famex.inc)

;-----
;
; CODE SECTION
;
;-----
;
; NAME      FastDiv
; RSEG      CRYPTO_CODE
;
; EXTRN     CODE(AddOps)
; EXTRN     CODE(ClearSegment)
; EXTRN     CODE(RunFmx_3)
;
; PUBLIC    _s_long_div
; PUBLIC    ?s_long_div
; PUBLIC    s_long_div
; PUBLIC    NormLength
;
;-----
; @func WORD | _s_long_div |
; computes a fast division of long integers. This function uses the
; <f BPv>div.d<ef> instruction of the <f Hp2>SMARTXA<ef> in order to compute the
; result of the division.
;
; This approach is not vulnerable like the conservative bit-shift computation.
; <Para DTBreak>Algorithm Flow
; <ex2>
; a' = (a_addr,a_length)
; if msb != 0 a' += 2**(a_length*32)
;
; (a_addr,a_length - b_length + 1) = quotient
;
; when a' is divided by (b_addr,b_length)
;
; (a_addr + a_length - b_length + 1,b_length) = remainder
;
; when a' is divided by (b_addr,b_length)
;
; "options.7" specifies "b" address space
;
; @rdesc The function returns a WORD value in R0
;
; @flag <gt; 0 | Return quotient length in DWORDS
; @flag 0 | if an error occurs
;
; @parm FXWORD | a_addr | address of dividend
; @parm BYTE | a_length | length of divisor
; @parm FXWORD | b_addr | address of divisor
; @parm BYTE | b_length | length of divisor
; @parm BOOL | msb | 0=msb not set, 1=msb set
; @parm BYTE | options | options flag

```

```

; @scan      0
; @USES      none
; @CHANGES  r0,r1,r2,r3,r4,r5,r6
; @CHANGES  HPTR1, XTOP1, XPTR1, XLEN1, YPTR1, YLEN1, RPTR1, MODE1
; @CHANGES  XPTR2, XLEN2, YPTR2, YLEN2, YPTR3, RPTR3, TOPP3, MODE3
; @CALLS     ENTER_ASM, c2r_1, c2_r3, NegB, CplB, RunFmx_1, RunFmx_3
; @FINALLY   EndFmx
;
;-----
UpdFmx      set    1      ; use 'correct' but expensive q+=r algorithm
TimeInVariant set    0      ; update quotient always, -> constant runtime

SaveOffs    SET     12
ahi         set     r1
dlo         set     r2
dhi         set     r3

;-----
; alen-blen+1
; | variable |-- cbAlen(real)---|--1--|--1--|
; | - quotient -| - a_length -| msb | ofs | - q -|--- b_length---|--1--|
; |          |          |          |          |          |
; cbQT      cbA          cbQ      cbB (if B was in Eeprom)
;-----
_s_long_div:
?s_long_div:
s_long_div:
    ENTER_ASM 7Eh
    xor.w    r0,r0          ; set rc=0
    push.w   r0             ; assume rc=0 (error by default)
;-----
; @Step 1 | copy b in EEPROM to FAMEX temp memory |
; The parameter b needs to be copied into temporary memory. First it will be
; verified, that the dividend has a length greater or equal to that of the divisor
;-----
    mov.b    r21,[r7+a_length] ; a_length < b_length ?
    cmp.b    r21,[r7+b_length] ; yes, error !
    bcc      _s_long_div_1
    jmp      EndFastDiv        ; rc = 0
;-----
; @cont Then the parameter b is taken from EEprom in order to be copied into
; FAMEX RAM
;-----
_s_long_div_1:
    add.b    r21,[r7+a_addr]    ; b_addr = a_addr+2+a_length
    adds.b   r21,#3             ; goto end of temp memory
; input to copy from EE ?
    mov.b    r11,[r7+options]   ; do we have b in EEPROM ?
    jnb      r11.7,noel         ;
    mov.b    RPTR3,r21          ; must not exceed 1 KB !
    mov.w    r0,[r7+b_addr]     ; #P:0002 length was wrong for EE(JWE)
    mov.b    [r7+b_addr],r21    ; fake b_addr to temp B RAM addr
    mov.b    HPTR3,r0h          ;
; @bug FIXED.SEQ:243 | this is an example bug, not a real one
    mov.b    r0h,[r7+b_length] ;
    call     FmxMemCpyEe        ; copy data from EE to temp space
noel:
;-----
; @step 2 | save temp vars |
; this saves the temp vars (nonsens comment - just an example)
;-----
    sub.w    r7,#SaveOffs      ;
...

```

...

```
mov.w    [r1+],[r0+]
djnz.w   r3,post2k
sub.w    r1,#FAMEXRAMBASE      ; Probably unnecessary
; -----
; @cont and finally ends the function
; -----
ret
END
```

## 6.11 Assembler Example output

The following represents the output generated from the Assembler example code

## 6.12 Module FASTDIV.AXA

Filename: F:\COMPS\PROJECTS\AD\TEST.AXA

### Version

Version 3 Driver 2

### Change History

**01-Nov-00, Version 1.0,**

**HS:** Module created

**13-Dec-00, Version 2.1,**

**HS:** Tempespace wasn't cleared correctly (P\_DIV\_01\_05)

**11-Jun-01, Version 3.0,**

**ml:** Convert to RIDE.

**06-Aug-01, Version 3.0,**

**je:** Convert from segment 15 offsets to physical addresses

**22-Aug-01, Version 3.0,**

**DCT:** Clean up include files

**18-Sep-01, Version 3.1,**

**HS:** Fixed save/restore of tempvars beyond 1KB

**04-Oct-01, Version 3.1,**

**je:** Add undecorated entry point

**30-Nov-01, Version 3.2,**

**DCT:** Change ROM\_CODE to CRYPTO\_CODE

**12-Mar-02, Version 4.0,**

**HS:** migration to Caernarvon kernel, AD tags added the \$-tags for the scan-hdr.exe function have been changed to ampersand tags in order to comply with the AUTODOC program

## 6.13 `_s_long_div`

### Function

computes a fast division of long integers. This function uses the *div.d* instruction of the **SMARTXA** in order to compute the result of the division.

This approach is not vulnerable like the conservative bit-shift computation.

#### Algorithm Flow

$a' = (a\_addr, a\_length)$  if  $msb \neq 0$   $a' += 2^{(a\_length \times 32)}$

$(a\_addr, a\_length - b\_length + 1) = \text{quotient}$

when  $a'$  is divided by  $(b\_addr, b\_length)$

$(a\_addr + a\_length - b\_length + 1, b\_length) = \text{remainder}$

when  $a'$  is divided by  $(b\_addr, b\_length)$

"options.7" specifies "b" address space

Defined in: F:\COMPS\PROJECTS\AD\TEST.AXA Line 71

### C Call

```
WORD _s_long_div(FXWORD a_addr,  
                 BYTE a_length,  
                 FXWORD b_addr,  
                 BYTE b_length,  
                 BOOL msb,  
                 BYTE options)
```

### Parameters

<i>a_addr</i>	address of dividend
<i>a_length</i>	length of divisor
<i>b_addr</i>	address of divisor
<i>b_length</i>	length of divisor
<i>msb</i>	0=msb not set, 1=msb set
<i>options</i>	options flag

### Return Codes

The function returns a WORD value in R0

> 0	Return quotient length in DWORDS
0	if an error occurs

### Bugs fixed

#### FIXED.SEQ:243

test.axa:Line 157: this is an example bug, not a real one

#### FIXED.SEQ:485

test.axa:Line 172: this is another example bug

### 6.13.1

#### **Step 1: copy b in EEPROM to FAMEX temp memory**

The parameter b needs to be copied into temporary memory. First it will be verified, that the dividend has a length greater or equal to that of the divisor

Then the parameter b is taken from EEprom in order to be copied into FAMEX RAM

### 6.13.2

#### **Step 2: save temp vars**

this saves the temp vars (nonsens comment - just an example)

and finally ends the function

## 6.14

# How to invoke AUTODOC

AUTODOC is a program that generates documentation from source code files. In order to provide code with the documentation an **AUTODOC preprocessor** is provided.

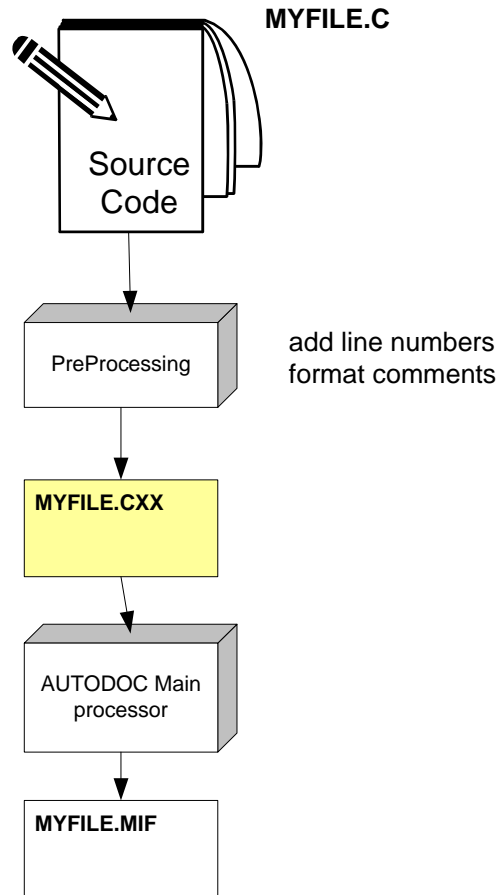


Figure 6-3. <Figure AutodocPhases>AUTODOC phases

The actual invocation of AUTODOC requires several parameters. For the use in the O/S kernel project, two batch files come in rather handy to make the use easier.

### **PrcAD.BAT** <filename> <extension>

processes <filename.ext>. Note, that there is no dot between the *filename* and the *extension*. This is in order to allow the batch file recognize the file type. Depending on the file type the pre-processor is invoked or not.

### **RunAD.BAT**

runs AUTODOC, however is not actually invoked by the user.



The following figure explains the file [PrcAD.BAT](#).

```
@echo off
    set AdName=%~n1
    set AdExt=%~x1
; Check extensions, pretestA for ASM, pretest for 'C'
    if "%AdExt%"=="." goto pretest
    if "%AdExt%"=="." goto pretest
    if "%AdExt%"=="." goto pretest
    if "%AdExt%"=="." goto pretest
    if "%AdExt%"=="." goto pretest
    if "%AdExt%"=="." goto pretestA
    if "%AdExt%"=="." goto pretestA
    if "%AdExt%"=="." goto pretestA
    if "%AdExt%"=="." goto pretestA
    if "%AdExt%"=="." goto pretestA
    if "%AdExt%"=="." goto pretestA
    if "%AdExt%"=="." goto pretestA
    if "%AdExt%"=="." goto pretestA
    if "%AdExt%"=="." goto pretestA

; Process AUTODOC without pre processing
:normProc
    call %PROSECCOS%\tools\bin\RunAd %AdName%AdExt% %2 %3 %4 %5 %6 %7
    goto PostProc

rem 'C' section
rem If a third parameter was given, process without pre processing
:pretest
    if "%2"==" " goto preinc
    goto normProc

:preinc
    call %PROSECCOS%\tools\bin\preinc %AdName%AdExt% /f
    call %PROSECCOS%\tools\bin\RunAd %AdName%.cxx
    goto PostProc

rem ASSEMBLER section
rem If a third parameter was given, process without pre processing
:pretestA
    if "%3"==" " goto preincA
    goto normProc

rem Process ASM files with preprocessor
:preincA
    call %PROSECCOS%\tools\bin\preinc %AdName%AdExt% /f
    call %PROSECCOS%\tools\bin\RunAd %AdName%.axx
    goto PostProc

:PostProc
; Delete temporary files and invoke FrameMaker (if you want)
; del %1.cxx
start %1.MIF
:ende
```

Figure 6-4. PrcAD.BAT

## 6.15 New Features

@Listfile invokation

how to use PDF links

---

# 7 Quick Guide for use with ID Workbench

---

## 7.1 Header Section

### @doc

Any document using the AUTODOC feature has to start with the **@doc** tag. Any other tags will not be interpreted unless the **@doc** tag was found in the document.

The **@doc** tag may be followed by a qualifier

**@doc** EXTERNAL will output the following definitions, if the EXTERNAL flag is defined at runtime of AD.EXE. @doc INTERNAL will output the subsequent definitions

### @module *name / description*

The **@module** tag describes the name and the function of the whole file. A module is an ASCII file containing one or more functions

The **@module** tag, when encountered in the source, lists several paragraphs in order to describe the version (**@version** tag) and change history (**@fix** tag) of the file.

### @version *version # / driver #*

The **@version** tag indicates the version.driver information of the module.

### @fix *data / version / author / description*

The **@fix** tag indicates the change history of the module. It shall be used in order to record changes to a file after its initial release

---

## 7.2 Code section

### 7.2.1 Paragraph tags

#### @func *type / name / desc*

The **@func** tag describes an assembler function in the layout of a typical 'C' function. It is typically followed by a list of parameters (**@parm**) and a description of the return values (**@rdesc / @flag**).

The *type* parameter describes the type of the output value of the function. This does not always apply to assembler functions. A possible solution to this dilemma could be using the MATHLAB notation of a vector output :

```
@func [int a, char* b] | TestFun | This function demonstrates multiple variable output
```

The generated type will follow this definition.

#### @parm *type / name / desc*

The **@parm** tag may appear multiple times, each time describing an input parameter of the function.

#### @output desc

The **@output** topic precedes a list of **@rc** tag that describe a particular set of values to be returned.

#### @rc *value / description*

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

### @vuln title | description

The **@vuln** is used to indicate a possible vulnerability in the code. Frequently programmers can identify potential vulnerabilities (timing attack) already on creating the code. The depth of leakage, however can only be investigated later. The **@vuln** tag can serve as warning to observe these parts in the code.

### @bug test reference | description

The **@bug** tag allows the description of bugs and fixes in the code. A list of bugs is collected for each function in order to receive an overview for the changes done to the code. The 'test reference' parameter allows to indicate a reference to the appropriate test scenario.

### @step Number | Header | description

The **@step** tag is used to comment a processing step.

### @section ID | Header | description

The **@section** tag is used to comment a processing step.

### @cont description

The **@cont** tag is used in order to continue the text of a preceding tag. It is formatted as 'Body' paragraph by default, however the paragraph format may be changed using the `<para [style] >` text tag.

The **@cont** text is often used to continue the description of a step in the code. After several statements you might want to describe what's going in the code. Using the **@cont** tag allows you to add the next comment and make it appear right after your previous comment given in a preceding **@step**. It will also continue other tags than **@step** only.

This tag is helpful to write your comment text located at the appropriate code statement. In the documentation the text keeps together with the text of the preceding (e.g. **@step**) tag.

### @catch description

The **@catch** tag is used in order to describe an error handler. The text in *description* describes the function of the **catch** section of a module.

## 7.2.2 Text tags

Text tags are used in order to change the formatting style within paragraphs

**<para [stylename] >**

sets the paragraph style. `<para BodyAfterHead>` makes the paragraph appear in this style.

**Note:** This tag is case sensitive and will not work if not written correctly.

**<f [fontname]>**

sets the current font. **<f BPv>** sets the "BPv" font, **<f Blue>** set the color blue until the **<ef>** tag or another paragraph is found.

**Note:** Do not set **<fBXmp...>** fonts. Use the following **<ex...>** tags instead.

Reason: The **<ex ..>** tags are more powerful and control the parser to take in the formatting of your file exactly without removing heading or trailing blanks. Only the first blank after the comment character (e.g. // or ; ) will be ignored.

**<ef>** Ends the font and returns to the normal font.

**<ex>** sets the BXmp font and avoids the removal of blanks in the beginning of a line. The paragraph style will be kept

**<ex1>** sets the BXmp paragraph and avoids the removal of blanks in the beginning of a line

<b>&lt;ex2&gt;</b>	sets the BXmp2 paragraph and avoids the removal of blanks in the beginning of a line
<b>&lt;ex3&gt;</b>	sets the BXmp3 paragraph and avoids the removal of blanks in the beginning of a line
<b>&lt;ex4&gt;</b>	sets the BXmp4 paragraph and avoids the removal of blanks in the beginning of a line
<b>&lt;fig <i>name</i>&gt;</b>	insert a figure anchor. See next paragraph
<b>&lt;rc&gt;</b>	sets the BPk font in order to describe a return code. Turned off with the <b>&lt;ef&gt;</b> tag.
<b>&lt;sv <i>name</i>&gt;</b>	insert a system variable
<b>&lt;tbl <i>name</i>&gt;</b>	insert a table anchor. See next paragraph
<b>&lt;xref <i>name</i>&gt;</b>	insert a cross reference anchor.
<b>&lt;v&gt;</b>	sets the BPv font in order to describe a variable. Turned off with the <b>&lt;ef&gt;</b> tag.

---

## 7.3 Special characters and symbols

Special characters may be inserted in two ways. In order to create any special ASCII code you may use the **^ prefix** to supply the two letter ASCII code.

This is in particular important using symbols. E.g. the greek letter  $\Phi$  can be generated by  
 $\Phi$  = <f Symbol>F<ef>

or with the prefix ^ using

$\Phi$  = <f Symbol>^46 <ef>

**Note:** It is essentially important to add a space character after the two hex nibbles. So the '46' in the example above is followed by a space before the **<ef>** tag ends the font.

The first option, is recommended, if the character has an ASCII value less than 128 ('80'x).

The second option is more appropriate, if the ASCII value of the character to be generated has an ASCII value greater than 127.

The ASCII values to directly correspond with the following table, taken from David Toll's famous BOOKMASTER guide which is added to this document as [Appendix A. "FrameMaker Character Sets" on page A-195.](#)

---

## 7.4 Figures and Tables

Another new feature has been added to the original AUTODOC program in order to support figures and tables in FrameMaker.

It is considered difficult to create a sophisticated figure or table layouts in ASCII text. Therefore another approach was taken.

**Figures and tables are created in a separate reference file <myname>.IDE.**

### 7.4.1 Step 1: Create the figures and tables in the reference file

You can link figures by reference in the `<myname>.fm` document. It is not necessary to imbed them. This has the advantage, that you can update your drawings without having to do the inclusion of the figure again.

In the reference file `<myname>.FM` the description part of a figure contains a special text identifier

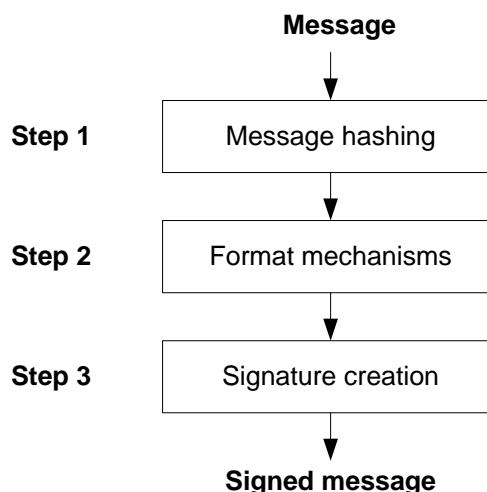


Figure 7-1. **<Figure MsgSign>** Signing a message with a SmartCard

The text identifier is formed in a **<Figure text>** construct.

A table is built in a similar way

Table 7-1. **<Table SignAct>** Signing Actions

Step	Action
Step 1	Message hashing
Step 2	Format Mechanisms
Step 3	Signature creation

Figures and tables such can be created entirely sophisticated and according to any choice of the editor.

### 7.4.2 Step 2: Export the reference file to MIF

Use the 'SaveAs' function of FrameMaker in order to export the `<myfile>.fm` as `<myfile>.MIF`. This avoids any tampering of the AUTODOC with the original `<myfile>.FM` and creates an ASCII readable (MIF) version of your figures.

In particular linked (by reference) figures by reference in the `.FM` document, will be exported as OLE objects in the MIF file. Such the MIF file contains the full drawing, even if the `.FM` file used the 'link by reference' option.

### 7.4.3 Step 3: Write the program source(s)

Use the `<fig name>` and `<tbl name>` text tags in order to insert any of your pre-designed figures into the final output document. name specifies the name that you gave a figure (table) in the caption.

Writing a text in the program (ASCII) file and using the `<fig name>/<tbl name>` text tag places the figure/table anchor exactly where the `<fig name>` or `<tbl name>` text tag was found.

The *name* variable specifies the same name as used in the FrameMaker <myfile>.FM caption of the appropriate figure.

The <fig *name*> in the program (ASCII) file corresponds to the <Figure *name*> entry in the FrameMaker <myfile>.FM file.

The <tbl *name*> in the program (ASCII) file corresponds to the <Table *name*> entry in the FrameMaker <myfile>.FM file.

The <xref *name*> in the program (ASCII) file corresponds to the <Xref *name*> entry in the FrameMaker <myfile>.FM file.

#### 7.4.4 Step 4: Run AUTODOC with the /b option

When invoking AUTODOC use the option flag /b in order to inform the AUTODOC program to use your figure reference file <myfile>.MIF.

A typical AUTODOC invocation could be ...

```
F:\ASM>AD /f "f:\fmt\fm.fmt" /b "..\REF\<myfile>" test.asm
```

The extension .MIF will automatically be added to the <myfile> if no extension is given explicitly.

AUTODOC merges the figures and tables, equations and cross references into the output file wherever you refer them.

**Note:** A figure or table shall be reference only once in one or a set of files processed with AUTODOC. If the same figure is referenced twice, FrameMaker will complain for double usage of a figure, when the .MIF output file is imported.

If you want to use the same figure twice, you must copy it in the reference document <myfile> and supply with a different <Figure ....> text tag. Hence you may refer the copy within your source. This effect is no limitation as in writing a FrameMaker document you wouldn't be able to use one figure in two places either.

---

## 7.5 Equations

The generation of an equation can be made with

$$\text{<eq TestEq1>Quota} = \text{sizeof(tMemBlock)} + \text{HdrSize} + \text{sizeof(IFINFO)} \quad (7-1)$$

in the reference document.

The source code will relate to this equation with the <eq TestEq1> text tag similar to the description of figures and tables above.

---

## 7.6 Updating figures, tables and equations

In many cases, flow charts and figures in the G&D OS are created with MicroSoft Visio 2000. The files <drawing>.VSD may be included as objects by reference into the FrameMaker Source. (File-Import-Object-Create From File-Link-Browse <\*.vsd>, do not forget to select the anchored frame previously created).

Changing such a <drawing>.VSD file will automatically update the FrameMaker .FM file, when opened next.

Whatever the way of creation is, after the reference file was brought up-to-date, in order to update the final output, you simply need to export the reference file again as <myfile>.MIF as discussed in 7.4.2 on page 7-158. Then run the AUTODOC program to create the final output source.

These few steps are considered affordable to support figures and tables without having too many tasks for the programmer.

Tables and Equations may be updated in the reference document directly.

---

## 7.7 Cross references

### 7.7.1 Cross references ex-source

If you want to generate a cross reference in your source to any external document, you need to use the <xref *name*> tag.

The reference document contains the actual reference, which is copied into the output document, when processing AUTODOC.

If the reference document contains

`<xref AsmExample>7.10 “Assembler Code Example” on page 7-161`

then the source code shall contain a statement

`; You may also refer to <xref AsmExample>`

During processing, AUTODOC will find the cross reference and insert it into the output document. Hence the output document will contain outbound cross references with a freshness of the reference document.

### 7.7.2 Cross references in-source

Cross references may be made to

- Headings
- Figures
- Tables
- Equations

AUTODOC prepares these paragraph styles with unique markers. The unique identifier of these markers is derived by a hash function from the heading text. Hence these markers will be exactly the same as long as the heading text (like a structure name etc.) is not changed. Also the heading of a **@step** statement is, for instance, part of this hash.

In order to generate a reference into an AUTODOC generated file, you simply have to make a cross reference into the appropriate .MIF or .FM file, depending what final file type you choose for your documentation

**Note:** We recommend the conversion of a generated .MIF file to .FM (File-SaveAs “.FM”). for backup security reasons. This way it is impossible by an accidental AUTODOC invocation to delete the actual documentation.

Typically FrameMaker alters the target document of reference, because it has to add a marker to the file. As AUTODOC has pre-generated all markers on headers, figures and tables, on establishing a cross reference to an AUTODOC generated file, the target file does not need to be changed. FrameMaker recognizes the existence of the marker and indeed does not alter the target file.

This, however has the effect, that ‘inbound’ cross references are constant and reliable unless the header title / tag fields are changed in the target document.



If this is the case, the cross reference for the altered tags would have to be updated. Therefore it is recommended to do the in-source cross references in a later stage of the source, where the tag descriptors are trustable.

---

## 7.8 System variables

Use the `<sv varname >` text tag to insert any system variable at any place in your text.

---

## 7.9 Adding system variables

You may list your personal additional system variables in the reference file. The global system variables defined in BOOKMSTR.MIF will be present anyway. So you would only have to enter your additional variables into the reference file. You may use the DT/DD tags like it is proposed here, however you should not use special fonts within the text, because this generates 'noise' while these variables are extracted from the reference file (in its MIF representation).

The additional system variables will be available in any output document.

```
<sv.Name>psmgr
      <sv.Def>persistent storage manager
```

---

## 7.10 Assembler Code Example

The following example demonstrates the typical tags used in Assembler files.

```
;*****
;
; Caernarvon Secure Operating System
;
; IBM Research Division, T. J. Watson Research Center
;
; (C) COPYRIGHT IBM CORP 2000, 2001, 2002
;
; LICENSED MATERIALS - PROPERTY OF IBM
;
; All Rights Reserved
;
; U.S. Government Users Restricted Rights - Use, duplication or
; disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
;
; See Copyright Instructions (Form G120-2083)
;
; Origins:          IBM T. J. Watson Research Center
;
; MODULE TYPE:      ASM Source File
;
; RESTRICTIONS:      DPA tests not yet finished
;
; CREATION DATE:     Created Nov 2000 by H. Scherzer, IBM Watson Research
;                    Center, Hawthorne
;
;*****
; @doc
; @module FASTDIV.AXA | implements the FAMEX FastDiv function
```

```

; @version 3 | 2
;
; @fix 01-Nov-00 | 1.0 | HS | Module created
; @fix 13-Dec-00 | 2.1 | HS | Temp space wasn't cleared correctly (P_DIV_01_05)
; @fix 11-Jun-01 | 3.0 | ml | Convert to RIDE.
; @fix 06-Aug-01 | 3.0 | je | Convert from segment 15 offsets to physical addresses
; @fix 22-Aug-01 | 3.0 | DCT | Clean up include files
; @fix 18-Sep-01 | 3.1 | HS | Fixed save/restore of tempvars beyond 1KB
; @fix 04-Oct-01 | 3.1 | je | Add undecorated entry point
; @fix 30-Nov-01 | 3.2 | DCT | Change ROM_CODE to CRYPTO_CODE
;
; @fix 12-Mar-02 | 4.0 | HS | migration to Caernarvon kernel, AD tags added
;                               the $-tags for the scanhdr.exe function have been
;                               changed to ampersand tags in order to comply with
;                               the AUTODOC program
;
; ***** End of Header *****
; $CASE
; $ZPAGE

$include (crypto.inc)
$include (famex.inc)

;-----
;
; CODE SECTION
;
;-----
;
; NAME      FastDiv
; RSEG      CRYPTO_CODE
;
; EXTRN     CODE(AddOps)
; EXTRN     CODE(ClearSegment)
; EXTRN     CODE(RunFmx_3)
;
; PUBLIC    _s_long_div
; PUBLIC    ?s_long_div
; PUBLIC    s_long_div
; PUBLIC    NormLength
;
;-----
; @func WORD | _s_long_div |
; computes a fast division of long integers. This function uses the
; <f BPv>div.d<ef> instruction of the <f Hp2>SMARTXA<ef> in order to compute the
; result of the division.
;
; This approach is not vulnerable like the conservative bit-shift computation.
; <Para DTBreak>Algorithm Flow
; <ex2>
; a' = (a_addr,a_length)
; if msb != 0 a' += 2**(a_length*32)
;
; (a_addr,a_length - b_length + 1) = quotient
;
; when a' is divided by (b_addr,b_length)
;
; (a_addr + a_length - b_length + 1,b_length) = remainder
;
; when a' is divided by (b_addr,b_length)
;
; "options.7" specifies "b" address space
;
; @rdesc The function returns a WORD value in R0
;
; @flag <gt; 0 | Return quotient length in DWORDS
; @flag 0      | if an error occurs

```

```

;
; @param FXWORD | a_addr | address of dividend
; @param BYTE | a_length | length of divisor
; @param FXWORD | b_addr | address of divisor
; @param BYTE | b_length | length of divisor
; @param BOOL | msb | 0=msb not set, 1=msb set
; @param BYTE | options | options flag

; @scan 0
; @USES none
; @CHANGES r0,r1,r2,r3,r4,r5,r6
; @CHANGES HPTR1, XTOP1, XPTR1, XLEN1, YPTR1, YLEN1, RPTR1, MODE1
; @CHANGES XPTR2, XLEN2, YPTR2, YLEN2, YPTR3, RPTR3, TOPP3, MODE3
; @CALLS ENTER_ASM, c2r_1, c2_r3, NegB, Cp1B, RunFmx_1, RunFmx_3
; @FINALLY EndFmx
;
;-----
UpdFmx set 1 ; use 'correct' but expensive q+=r algorithm
TimeInVariant set 0 ; update quotient always, -> constant runtime

SaveOffs SET 12
ahi set r1
dlo set r2
dhi set r3

;-----
; alen-blen+1
; | variable |-- cbAlen(real)---|--1--|--1--|
; | - quotient -| - a_length -| msb | ofs | - q -| --- b_length---|--1--|
; |
; cbQT cbA cbQ cbB (if B was in Eeprom)
;-----
_s_long_div:
?s_long_div:
s_long_div:
ENTER_ASM 7Eh
xor.w r0,r0 ; set rc=0
push.w r0 ; assume rc=0 (error by default)

;-----
; @Step 1 | copy b in EEPROM to FAMEX temp memory |
; The parameter b needs to be copied into temporary memory. First it will be
; verified, that the dividend has a length greater or equal to that of the divisor
;-----
mov.b r21,[r7+a_length] ; a_length < b_length ?
cmp.b r21,[r7+b_length] ; yes, error !
bcc _s_long_div_1
jmp EndFastDiv ; rc = 0

;-----
; @cont Then the parameter b is taken from EEPROM in order to be copied into
; FAMEX RAM
;-----
_s_long_div_1:
add.b r21,[r7+a_addr] ; b_addr = a_addr+2+a_length
adds.b r21,#3 ; goto end of temp memory
; input to copy from EE ?
mov.b r11,[r7+options] ; do we have b in EEPROM ?
jnb r11.7,noel ;
mov.b RPTR3,r21 ; must not exceed 1 KB !
mov.w r0,[r7+b_addr] ; #P:0002 length was wrong for EE(JWE)
mov.b [r7+b_addr],r21 ; fake b_addr to temp B RAM addr
mov.b HPTR3,r0h ;
; @bug FIXED.SEQ:243 | this is an example bug, not a real one
mov.b r0h,[r7+b_length] ;
call FmxMemCpyEe ; copy data from EE to temp space
noel:
;-----

```

```

; @step 2 | save temp vars |
; this saves the temp vars (nonsens comment - just an example)
; -----
        sub.w    r7,#SaveOffs        ;

...

...

        mov.w    [r1+],[r0+]
        djnz.w   r3,post2k
        sub.w    r1,#FAMEXRAMBASE    ; Probably unnecessary
; -----
; @cont and finally ends the function
; -----
        ret
END

```

---

## 7.11 Assembler Example output

The following represents the output generated from the Assembler example code

---

## 7.12 Module FASTDIV.AXA

Filename: F:\COMPS\PROJECTS\AD\TEST.AXA

### Version

Version 3 Driver 2

### Change History

- 01-Nov-00, Version 1.0,**  
HS: Module created
- 13-Dec-00, Version 2.1,**  
HS: Tempspace wasn't cleared correctly (P\_DIV\_01\_05)
- 11-Jun-01, Version 3.0,**  
ml: Convert to RIDE.
- 06-Aug-01, Version 3.0,**  
je: Convert from segment 15 offsets to physical addresses
- 22-Aug-01, Version 3.0,**  
DCT: Clean up include files
- 18-Sep-01, Version 3.1,**  
HS: Fixed save/restore of tempvars beyond 1KB
- 04-Oct-01, Version 3.1,**  
je: Add undecorated entry point
- 30-Nov-01, Version 3.2,**  
DCT: Change ROM\_CODE to CRYPTO\_CODE

## 7.13 \_s\_long\_div

### Function

computes a fast division of long integers. This function uses the *div.d* instruction of the **SMARTXA** in order to compute the result of the division.

This approach is not vulnerable like the conservative bit-shift computation.

#### Algorithm Flow

$a' = (a\_addr, a\_length)$  if  $msb \neq 0$   $a' += 2^{(a\_length \times 32)}$

$(a\_addr, a\_length - b\_length + 1) = \text{quotient}$

when  $a'$  is divided by  $(b\_addr, b\_length)$

$(a\_addr + a\_length - b\_length + 1, b\_length) = \text{remainder}$

when  $a'$  is divided by  $(b\_addr, b\_length)$

"options.7" specifies "b" address space

Defined in: F:\COMPS\PROJECTS\AD\TEST.AXA Line 71

### C Call

```
WORD _s_long_div(FXWORD a_addr,
                BYTE a_length,
                FXWORD b_addr,
                BYTE b_length,
                BOOL msb,
                BYTE options)
```

### Parameters

<i>a_addr</i>	address of dividend
<i>a_length</i>	length of divisor
<i>b_addr</i>	address of divisor
<i>b_length</i>	length of divisor
<i>msb</i>	0=msb not set, 1=msb set
<i>options</i>	options flag

### Return Codes

The function returns a WORD value in R0

<b>&gt; 0</b>	Return quotient length in DWORDS
<b>0</b>	if an error occurs

## Bugs fixed

### **FIXED.SEQ:243**

test.axa:Line 157: this is an example bug, not a real one

### **FIXED.SEQ:485**

test.axa:Line 172: this is another example bug

## **7.13.1 Step 1: copy b in EEPROM to FAMEX temp memory**

The parameter b needs to be copied into temporary memory. First it will be verified, that the dividend has a length greater or equal to that of the divisor

Then the parameter b is taken from EEprom in order to be copied into FAMEX RAM

## **7.13.2 Step 2: save temp vars**

this saves the temp vars (nonsens comment - just an example)

and finally ends the function

## 8 Adobe SDK license agreement.

### ADOBE ACROBAT SOFTWARE DEVELOPER'S KIT SOFTWARE LICENSE AGREEMENT

NOTICE TO USER: PLEASE READ THIS CONTRACT CAREFULLY. BY USING ALL OR ANY PORTION OF THE SOFTWARE YOU ACCEPT ALL THE TERMS AND CONDITIONS OF THIS AGREEMENT, INCLUDING, IN PARTICULAR, THE LIMITATIONS ON: USE CONTAINED IN SECTIONS 2, 3, AND 4; WARRANTY IN SECTION 7; LIABILITY IN SECTION 8, AND SPECIFIC LIMITATIONS IN SECTION 14. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU. IF YOU DO NOT AGREE, DO NOT USE THIS SOFTWARE. IF YOU ACQUIRED THE SOFTWARE ON TANGIBLE MEDIA (e.g., CD-ROM) WITHOUT AN OPPORTUNITY TO REVIEW THIS LICENSE AND YOU DO NOT ACCEPT THIS AGREEMENT, YOU MAY NOT USE THE SOFTWARE.

Adobe and its suppliers own all intellectual property in the Software. Adobe permits You to Use the Software only in accordance with the terms of this Agreement. Use of some third-party materials included in the Software may be subject to other terms and conditions typically found in a separate license agreement or ReadMe file located near such materials.

## 1. DEFINITIONS

**1.1 "Adobe"** means Adobe Systems Incorporated, a Delaware corporation, 345 Park Avenue, San Jose, California 95110, if Section 11(a) of this Agreement applies; otherwise it means Adobe Systems Software Ireland Limited, Unit 3100, Lake Drive, City West Campus, Saggart D24, Republic of Ireland, a company organized under the laws of Ireland and an affiliate and licensee of Adobe Systems Incorporated.

**1.2 "Adobe Reader Products"** means Adobe® Reader®, Adobe Acrobat® Reader, Adobe Acrobat Approval, Adobe Acrobat Elements, and related viewer products.

**1.3 "Developer," "You," and "Your"** refer to any person or entity Using this Software, or any component thereof.

**1.4 "Developer Programs"** means Your application programs that are designed to function with Other Adobe Acrobat Software products.

**1.5 "Documentation"** means explanatory materials supplied with the Software or made available online on Adobe public Web pages related to the Software.

**1.6 "End User License Agreement"** means an end user license agreement that provides a:

- a. (a) limited, nonexclusive right to use the subject Developer Program with no further right to reproduce (except for archival and/or backup copies permitted by law) and/or distribute the subject Developer Program;
- b. (b) prohibition against distributing, selling, sublicensing, renting, loaning, or leasing the subject Developer Program;
- c. (c) prohibition against reverse engineering, decompiling, disassembling, or otherwise attempting to discover the source code of the subject Developer Program that is substantially similar to that set forth in Section 3 below;
- d. (d) statement that, if Your customer requires any Other Adobe Acrobat Software in order to use the Developer Program,
  - i. (i) Your customer must obtain such Other Adobe Acrobat Software via a valid license, and
  - ii. (ii) Your customer's use of such Other Adobe Acrobat Software must be in accordance with the terms and conditions of the end user license

agreement that ships with such Other Adobe Acrobat Software;

- e. (e) statement that You and Your suppliers retain all right, title, and interest in the subject Developer Program that is substantially similar to that set forth as Section 5 below;
- f. (f) statement that Your suppliers disclaim all warranties, conditions, representations, or terms with respect to the subject Developer Program substantially similar to the disclaimer set forth as Section 7 below; and
- g. (g) limit of liability substantially similar to that set forth as Section 8 below for the benefit of Your suppliers.

**1.7 "Header File Information"** means any header files (including but not limited to \*.h files) supplied in connection with the Software, including without limitation any related information detailing contents of header files.

**1.8 "Other Adobe Acrobat Software"** means the generally commercially available versions of Adobe Acrobat Standard, Adobe Acrobat Professional, and Adobe Reader Products.

**1.9 "Redistributable Code"** means certain object code files designated in the Documentation as "Redistributable Code."

**1.10 "Sample Code"** means sample software in source code format designated in the Documentation as "Sample Code" and/or "Snippets."

**1.11 "Software"** means all of the contents of the files, disk(s), CD-ROM(s) or other media with which this Agreement is provided, including but not limited to

- i. (i) Sample Code;
- ii. (ii) Header File Information;
- iii. (iii) Redistributable Code;
- iv. (iv) Documentation;
- v. (v) Software Tools; and
- vi. (vi) any upgrades, modified versions, updates, and/or additions thereto, if any, provided to You by Adobe.

**1.12 "Software Tools"** means developer tools (including but not limited to plug-ins and compiled samples) supplied with the Software, the selection of which may change from time to time at Adobe's sole discretion.

**1.13 "Use", "Used" or "Using"** means to access, install, download, copy, or otherwise benefit from using the Software.

## 2. LICENSE

Subject to the terms and conditions of this Software License Agreement (this "Agreement"), Adobe grants You a non-exclusive, nontransferable, royalty-free license to

- a. (a) Use the Software for the **sole purpose of internally developing Developer Programs**,
- b. (b) **reproduce and modify Sample Code** as a component of Developer Programs that add significant and primary functionality to the Sample Code,
- c. (c) **reproduce Redistributable Code** solely as a component of Developer Programs that add significant and primary functionality to the Redistributable Code, and



- d. (d) distribute Sample Code and/or Redistributable Code **in object code form only** as a component of Developer Programs that add significant and primary functionality to the Sample Code and/or Redistributable Code, provided that
  - i. (i) **You distribute such object code** under the terms and conditions of an End User License Agreement,
  - ii. (ii) You include a copyright notice reflecting the copyright ownership of Developer in such Developer Programs,
  - iii. (iii) You shall be solely responsible **to Your customers** for any update or support obligation or other liability which may arise from such distribution,
  - iv. (iv) You shall not make any statements that Your Developer Product is "certified," or that its performance is guaranteed, by Adobe, and
  - v. (v) You do not use Adobe's name or trademarks to market Your Developer Product without written permission of Adobe. Any modified or merged portion of the Sample Code, and/or merged portion of the Redistributable Code, IS subject to this Agreement. Use of Other Adobe Acrobat Software and/or any additional Adobe application program is subject to the applicable end user license agreement for such application software, even if such Other Adobe Acrobat Software or additional Adobe application program is supplied to You in connection with this Agreement. **You may make a limited number of copies of the Documentation to be used by Your employees or consultants for internal development purposes and not for general business purposes** or for distribution by any means, **and such employees or consultants shall be subject to this Agreement**. You may distribute Other Adobe Acrobat Software with Your Developer Programs only under separate license from Adobe. Adobe is under no obligation to provide any support under this Agreement, including upgrades or future versions of the Software, Other Adobe Acrobat Software and/or any component thereof, to Developer, end users, or to any other party. **Further developer support, software licensing, trademark licensing and trademark usage information is available through <http://www.adobe.com> and/or <http://partners.adobe.com>.**

### 3. RESTRICTIONS

**3.1 General Restrictions.** Except for the limited distribution rights as provided in Section 2 above with respect to Sample Code and Redistributable Code, You may not distribute, sell, sublicense, rent, loan, or lease the Software, Other Adobe Acrobat Software, and/or any component thereof to any third party. You also agree not to add or delete any program files that would modify the functionality and/or appearance of Other Adobe Acrobat Software and/or any component thereof. You agree not to reverse engineer, decompile, disassemble or otherwise attempt to discover the source code of the Software, Other Adobe Acrobat Software and/or any component thereof except to the extent (i) You may be expressly permitted to decompile under applicable law, (ii) it is essential to do so in order to achieve operability of the Software or Other Adobe Acrobat Software with another software program, and (iii) You have first asked Adobe to provide the information necessary to achieve such operability and Adobe has not made such information available. Adobe has the right to impose reasonable conditions and to request a reasonable fee before providing such information. Any such information supplied by Adobe and any information obtained by You by such permitted decompilation may only be used by

You for the purpose described herein and may not be disclosed to any third party or used to create any software which is substantially similar to the expression of the Software and/or Other Adobe Acrobat Software. Requests for such information should be directed to the Adobe Customer Support Department.

**3.2 Development Restrictions.** You agree that **You will not Use the Software to create, develop or use any program, software or service which**

- a. (a) removes the menu item that calls up the "About Screen" in any Other Adobe Acrobat Software product, other product incorporating Other Adobe Acrobat Software under valid license from Adobe, or any component thereof;
- b. (b) can both
  - i. (i) communicate with Adobe Reader Products and
  - ii. (ii) modify or save a PDF (Portable Document Format) document (including saving any modifications to a separate file for such documents);
  - iii. (c) exposes and/or discloses Header File Information;
- c. (d) works as a plug-in with Adobe Reader Products, unless specifically licensed to do so by Adobe;**
- d. (e) opens encrypted documents without the authorized knowledge of the document passwords or violates the access rights specified for a document;
- e. (f) provides access and/or displays content secured using digital rights management services or technology unless the Developer Product meets certain certification criteria in accordance with Adobe's then current certification process;
- f. (g) modifies or replaces the digital signature validation functionality and/or capability of Other Adobe Acrobat Software without written approval from Adobe;
- g. (h) validates digitally signed documents (including without limitation, adding any trust capability to documents) without written approval from Adobe;
- h. (i) modifies the permissions or rights in a PDF file enabled using Adobe enabling technology;
- i. (j) enables Other Adobe Acrobat Software to run on a server;
- j. (k) contains any viruses, Trojan horses, worms, time bombs, cancelbots or other computer programming routines that are intended to damage, detrimentally interfere with, surreptitiously intercept or expropriate any system, data or personal information;
- k. (l) when used in the manner in which it is intended, violates any material law, statute, ordinance or regulation (including without limitation the laws and regulations governing export control, unfair competition, antidiscrimination or false advertising); or
- l. (m) interferes with the operability of other Adobe or thirdparty programs or software which run with Other Adobe Acrobat Software.

#### **4. CONFIDENTIAL INFORMATION**

You agree not to disseminate or in any way disclose Header File Information to any person, firm or business except for Your employees who need to know such Header File Information and who have previously agreed to be bound by a confidentiality obligation consistent with the obligation set forth in this Section 4. Further, You agree to treat the Header File Information with the same degree of care as You accord to Your own confidential information, but in any event no less than reasonable care. Your obligations under this section with respect to the Header File Information shall terminate when You can document that such Header File Information was

- i. (i) in the public domain at or subsequent to the time it was communicated to You by Adobe through no fault of Yours,
- ii. (ii) developed by Your employees or agents independently of and without reference to any information communicated to You by Adobe; or (iii) disclosed in response to a valid order by a court or other governmental body, as otherwise required by law, or as necessary to establish the rights of either party under this Agreement.

## 5. INTELLECTUAL PROPERTY, OWNERSHIP, COPYRIGHT PROTECTION

The Software and any authorized copies that You make are the intellectual property of and are owned by Adobe and its suppliers. The structure, organization and code of the Software are the valuable trade secrets and confidential information of Adobe and its suppliers. The Software is protected by law, including without limitation the copyright laws of the United States and other countries, and by international treaty provisions. You agree to protect Adobe's copyright and other ownership interests in all items in the Software. You agree that all copies of items in the Software reproduced for any reason by You will contain the same copyright, trademark, and other proprietary notices as appropriate and appear on or in the master items delivered by Adobe in the Software. Adobe and/or its suppliers retain all right, title and ownership throughout the world in the intellectual property embodied within the Software. Except as stated herein, this Agreement does not grant You any rights to patents, copyrights, trade secrets, trademarks, or any other rights in respect to the items in the Software, and all rights not expressly granted are reserved by Adobe and its suppliers.

## 6. TERM

This Agreement is effective until terminated. Adobe has the right to terminate this Agreement immediately if You fail to comply with any term of this Agreement. Upon any such termination, You must

- a. (a) return all full and partial copies of the items in the Software immediately to Adobe and
- b. (b) discontinue distribution of any Sample Code and/or Redistributable Code. Sections 1, 3, 4, 5, 6, 7, 8, 9, 11, 12 and 14 shall survive any termination and/or expiration of this Agreement.

## 7. DISCLAIMER OF WARRANTY

Adobe licenses the Software to You on an "AS IS" basis and without warranty of any kind. ADOBE AND ITS SUPPLIERS DO NOT AND CANNOT WARRANT THE PERFORMANCE OR RESULTS YOU MAY OBTAIN BY USING THE SOFTWARE. EXCEPT FOR ANY WARRANTY, CONDITION, REPRESENTATION OR TERM TO THE EXTENT TO WHICH THE SAME CANNOT OR MAY NOT BE EXCLUDED OR LIMITED BY LAW APPLICABLE TO YOU IN YOUR JURISDICTION, ADOBE AND ITS SUPPLIERS MAKE NO WARRANTIES, CONDITIONS, REPRESENTATIONS OR TERMS, EXPRESS OR IMPLIED, WHETHER BY STATUTE, COMMON LAW, CUSTOM, USAGE OR OTHERWISE AS TO THE SOFTWARE OR ANY COMPONENT THEREOF, INCLUDING BUT NOT LIMITED TO NONINFRINGEMENT OF THIRD PARTY RIGHTS, INTEGRATION, MERCHANTABILITY, SATISFACTORY QUALITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Some states or provinces do not allow the exclusion of implied warranties so the above limitations may not apply to You. You may have rights that vary from jurisdiction to jurisdiction.

For further warranty information, You may contact the Adobe Solutions Network at the Adobe Systems Incorporated address provided above.

## 8. LIMITATION OF LIABILITY

IN NO EVENT WILL ADOBE OR ITS SUPPLIERS BE LIABLE TO YOU FOR ANY DAMAGES, CLAIMS OR COSTS WHATSOEVER ARISING FROM THIS AGREEMENT AND/OR YOUR USE OF THE SOFTWARE OR ANY COMPONENT THEREOF, INCLUDING WITHOUT LIMITATION ANY CONSEQUENTIAL, INDIRECT, INCIDENTAL DAMAGES, OR ANY LOST PROFITS OR LOST SAVINGS, EVEN IF AN ADOBE REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSS, DAMAGES, CLAIMS OR COSTS OR FOR ANY CLAIM BY ANY THIRD PARTY. THE FOREGOING LIMITATIONS AND EXCLUSIONS APPLY TO THE EXTENT PERMITTED BY APPLICABLE LAW IN YOUR JURISDICTION. ADOBE'S AGGREGATE LIABILITY AND THAT OF ITS SUPPLIERS UNDER OR IN CONNECTION WITH THIS AGREEMENT SHALL BE LIMITED TO FIFTY U.S. DOLLARS (\$50.00). Nothing contained in this Agreement limits Adobe's liability to You in the event of death or personal injury resulting from Adobe's negligence or for the tort of deceit (fraud). Adobe is acting on behalf of its suppliers for the purpose of disclaiming, excluding and/or limiting obligations, warranties and liability as provided in this Agreement, but in no other respects and for no other purpose.

## 9. INDEMNIFICATION

You agree to defend, indemnify, and hold Adobe and its suppliers harmless from and against any claims or lawsuits, including attorneys' reasonable fees, that arise or result from the use or distribution of Developer Programs, provided that Adobe gives You prompt written notice of any such claim, tenders to You the defense or settlement of such a claim at Your expense, and cooperates with You, at Your expense, in defending or settling such claim.

## 10. GOVERNMENT REGULATIONS

You agree that any Developer Program that includes Sample Code and/or Redistributable Code (i) will include in its license agreement a reference to applicable U.S. Government regulations which control licensing of software and (ii) will not be shipped, transferred, or exported into any country or Used in any manner prohibited by the United States Export Administration Act or any other export laws, restrictions or regulations (collectively the "Export Laws"). In addition, if any part of the Software is identified as export controlled items under the Export Laws, You represent and warrant that You are not a citizen, or otherwise located within, an embargoed nation (including without limitation Iran, Iraq, Syria, Sudan, Libya, Cuba, North Korea and Serbia) and that You are not otherwise prohibited under the Export Laws from receiving the Software. All rights to Use the Software are granted on condition that such rights are forfeited if You fail to comply with the terms of this Agreement.

## 11. GOVERNING LAW

This Agreement will be governed by and construed in accordance with the substantive laws in force: (a) in the State of California, if a license to the Software is obtained when You are in the United States, Canada, or Mexico; or (b) in Japan, if a license to the Software is obtained when You are in Japan, China, Korea, or other Southeast Asian country where all official languages are written in either an ideographic script (e.g., hanzi, kanji, or hanja), and/or other script based upon or similar in structure to an ideographic script, such as hangul or kana; or (c) Ireland, if a license to the Software is obtained when You are in any other jurisdiction not described above. The respective courts of Santa Clara County, California when California law applies, Tokyo District Court in Japan, when Japanese law applies, and the courts of Ireland, when the law of Ireland applies, shall each have non-exclusive jurisdiction over all disputes relating to this Agreement. This Agreement will not be governed by the conflict of law rules of any jurisdiction or the United Nations Convention on Contracts for the International Sale of Goods, the application of which is expressly excluded.

## 12. GENERAL

You may not assign Your rights or obligations granted under this Agreement without the prior written consent of Adobe. None of the provisions of this Agreement shall be deemed to have been waived by any act or acquiescence on the part of Adobe, its agents, or employees, but only by an instrument in writing signed by an authorized signatory of Adobe. It is expressly agreed that a breach of Section 3 or 4 of this Agreement will cause irreparable harm to Adobe and that a remedy at law will be inadequate. Therefore, in addition to any and all remedies available at law, Adobe will be entitled to seek an injunction or other equitable remedies in all legal proceedings in the event of any threatened or actual violation thereof. When conflicting language exists between this Agreement and any other agreement included in this Software (except for the Integration Key License Agreement or any agreement supplied with Other Adobe Acrobat Software), this Agreement shall supersede. If either Adobe or Developer employs attorneys to enforce any rights arising out of or relating to this Agreement, the prevailing party shall be entitled to recover reasonable attorneys' fees. You acknowledge that You have read this Agreement, understand it, and that it is the complete and exclusive statement of Your agreement with Adobe which supersedes any prior agreement, oral or written, between Adobe and You with respect to the licensing to You of the Software. No

variation of the terms of this Agreement will be enforceable against Adobe unless Adobe gives its express consent in a writing signed by an authorized signatory of Adobe. If any part of this Agreement is found void and unenforceable, it will not affect the validity of the balance of this Agreement, which shall remain valid and enforceable according to its terms. The English version of this Agreement shall be the version used when interpreting or construing this Agreement.

## 13. NOTICE TO U.S. GOVERNMENT END USERS

**13.1 The Software and Documentation are "Commercial Item(s),"** as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users

- a. (a) only as Commercial Items and
- b. (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA.

**13.2 U.S. Government Licensing of Adobe Technology.** You agree that when licensing the Software for acquisition by the U.S. Government, or any contractor therefore, You will license consistent with the policies set forth in 48 C.F.R. §12.212 (for civilian agencies) and 48 C.F.R. §§227-7202-1 and 227-7202-4 (for the Department of Defense). For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference in this

Agreement.

## 14. SPECIFIC EXCEPTIONS

**14.1 Limited Warranty for Users Residing in Germany or Austria.** If You obtained the Software in Germany or Austria, and You usually reside in such country, then Section 7 does not apply, instead, Adobe warrants that the Software provides the functionalities set forth in the Documentation (the "agreed upon functionalities") for the limited warranty period following receipt of the Software when Used on the recommended hardware configuration. As used in this Section, "limited warranty period" means one (1) year if You are a business user and two (2) years if You are not a business user. Non-substantial variation from the agreed upon functionalities shall not be considered and does not establish any warranty rights. THIS LIMITED WARRANTY DOES NOT APPLY TO SOFTWARE PROVIDED TO YOU FREE OF CHARGE, FOR EXAMPLE, UPDATES, PRE-RELEASE, TRYOUT, PRODUCT SAMPLER, NOT FOR RESALE (NFR) COPIES OF SOFTWARE, OR SOFTWARE THAT HAS BEEN ALTERED BY YOU, TO THE EXTENT SUCH ALTERATIONS CAUSED A DEFECT. To make a warranty claim, during the limited warranty period You must return, at our expense, the Software and proof of purchase to the location where You obtained it. If the functionalities of the Software vary substantially from the agreed upon functionalities, Adobe is entitled, by way of re-performance and at its own discretion, to repair or replace the Software. If this fails, You are entitled to a reduction of the purchase price (reduction) or to cancel the purchase agreement (rescission). For further warranty information, please contact Adobe's Customer Support Department.

### **14.2 Limitation of Liability for Users Residing in Germany and Austria.**

**14.2.1** If You obtained the Software in Germany or Austria, and You usually reside in such country, then Section 8 does not apply. Instead, subject to the provisions in Section 14.2.2, Adobe's statutory liability for damages shall be limited as follows:

- i. (i) Adobe shall be liable only up to the amount of damages as typically foreseeable at the time of entering into the purchase agreement in respect of damages caused by a slightly negligent breach of a material contractual obligation and
- ii. (ii) Adobe shall not be liable for damages caused by a slightly negligent breach of a non-material contractual obligation.

**14.2.2** The aforesaid limitation of liability shall not apply to any mandatory statutory liability, in particular, to liability under the German Product Liability Act, liability for assuming a specific guarantee or liability for culpably caused personal injuries.

**14.2.3** You are required to take all reasonable measures to avoid and reduce damages, in particular to make back-up copies of the Software and Your computer data subject to the provisions of this Agreement.

**14.3 Pre-release Product Additional Terms.** If the product You have received with this license is a pre-commercial release or beta Software ("Pre-release Software"), then the following Section applies. To the extent that any provision in this Section is in conflict with any other term(s) or condition(s) in this Agreement, this Section shall supercede such other term(s) and condition(s) with respect to the Pre-release Software, but only to the extent necessary to resolve the conflict. You acknowledge that the Software is a pre-release version, does not represent final product from Adobe, and may contain bugs, errors and other problems that could cause system or other failures and data loss. Consequently, the Pre-release Software is provided to You "AS-IS", and Adobe disclaims any warranty or liability obligations to You of any kind. WHERE LIABILITY CANNOT BE EXCLUDED FOR PRE-RELEASE SOFTWARE, BUT IT MAY BE LIMITED, ADOBE'S LIABILITY AND THAT OF ITS SUPPLIERS SHALL BE LIMITED TO THE SUM OF FIFTY DOLLARS (U.S. \$50) IN TOTAL. You acknowledge that Adobe has not promised or guaranteed to You that Pre-release Software will be announced or made available to anyone in the future, Adobe has no express or implied obligation to You to announce or introduce the Pre-release Software and that Adobe may not introduce a product similar to or compatible with the Pre-release Software. Accordingly, You acknowledge that any research or development that You perform regarding the Pre-release Software

or any product associated with the Prerelease Software is done entirely at Your own risk. During the term of this Agreement, if requested by Adobe, You will provide feedback to Adobe regarding testing and use of the Pre-release Software, including error or bug reports. If You have been provided the Pre-release Software pursuant to a separate written agreement, such as the Adobe Systems Incorporated Serial Agreement for Unreleased Products, Your Use of the Software is also governed by such agreement.

You agree that You may not and certify that You will not sublicense, lease, loan, rent, assign, or transfer the Pre-release Software. Upon receipt of a later unreleased version of the Pre-release Software or release by Adobe of a publicly released commercial version of the Software, whether as a stand-alone product or as part of a larger product, You agree to return or destroy all earlier Pre-release Software received from Adobe and to abide by the terms of the license agreement for any such later versions of the Pre-release Software. Notwithstanding anything in this Section to the contrary, if You are located outside the United States of America, You agree that You will return or destroy all unreleased versions of the Pre-release Software within thirty (30) days of the completion of Your testing of the Software when such date is earlier than the date for Adobe's first commercial shipment of the publicly released (commercial) Software.

If You have any questions regarding this Agreement or if You wish to request any information from Adobe please use the address and contact information included with this product to contact the Adobe office serving Your jurisdiction. Adobe, Acrobat, and Reader are either trademarks or registered trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Acrobat SDK English 02.11.04





---

## 9 AUTODOC Design

HS\_23Mar06: Untersuchungen ergaben ...

---

### 9.1 Named Destinations

FrameMaker: Hypertext - Set Named Destination - erlaubt das Setzen eines Markers. Das hat Vorteile bei der Auflösung. In xref document kann man dann mit "Hypertext - Open Document" eine **NamedDestination** angeben, welche dann in das Target auflöst.

---

### 9.2 Named Destinations versus Cross References

FrameMaker CrossReferences haben den **Vorteil**, daß sie einen

- automatischen Update auf "Page" - "Volume" - "Chapter#" haben. Das kann man bereits im FrameMaker direkt ansteuern
- über das Cross Reference Format sehr elegant den Reference Text einstellen können.
- Cross Referenzen zur runtime-erstellung des Dokumente aufgelöst werden.

Allerdings muss man dabei Folgendes beachten:

- Die Generierung von PDF Files muss "von Buch zu Buch" passieren. Das heisst, ich muss ein Book drucken und während dieses Befehls müssen **alle Books**, zu welchen meine **Target Files** gehören, ebenfalls geöffnet sein !

**Nachteile** der CrossReferences sind...

- Cross References werden vom Distiller in der PDF auch als NamedDestinations angelegt. Allerdings wird dann eine Konstruktion `M<fileNo>.9.<hash> <name>` benutzt, bei der insbesondere die `fileNo` kritisch ist. Erzeugt man eine Cross Reference, dann ist dieser Marker bereits bekannt. AutoDoc dagegen kennt bei der Generierung noch nicht das Book, in welchem die File später reinkommt. AutoDoc kennt nicht mal die Directory, wo das Target File später reingelegt wird

Deshalb hat AutoDoc sicher ein Problem, wenn es eine Cross Referenz direkt erzeugen will. Mit der Brücke über SECCOSREF.FM passiert das deswegen nicht, weil mit SECCOSREF.FM das Target Book bereits geöffnet ist.

- Ein Problem besteht darin, daß eine File nicht an den Ort der Source generiert wird. Die Kenntnis der SourceFileDir hilft also nicht, um Kenntnis über den Ort der generierten File zu bekommen. Deshalb kann man nicht während der Generierung schon die Marker korrekt setzen
- Mit M8. **NamedDestinations** kann der Name der PDF Datei über **Prefix** gefunden werden, allerdings kann man dann keine Sprünge mehr direkt in der FrameMaker Source machen.

---

### 9.3 Idee

Wahrscheinlich muss ich die Marker in NamedDestination Marker ändern. Damit verliere ich die automatische Titelgenerierung

Von FrameMaker nach AutoDoc ist das kein Problem, denn da kann ich weiterhin mit Cross References arbeiten -> **Die M9 Marker sollten bleiben !!!!!**

Von AutoDoc nach AutoDoc sollten allerdings zusätzlich M8 Marker entstehen, die dann eine Named Destination mit dem xref\_Text darstellen.

Bei der CrossReferenzierung werden im xref\_Text nunmehr auch Kürzel angegeben.

<xref API\_qTlvFindObject> anstatt <xref qTlvFindObject>

Damit kann man kontrollieren, auf welches Target man geht. Gemäß der alten Konvention wird dann API umgesetzt in die API.PDF.

AutoDoc lernt die Marker über die normale Plug-In Export Funktion. In dem eigentlichen **Target (!)** ist die **NamedDestination nicht mit dem Prefix angeben**, das setzt AutoDoc richtig um.

ToDo .... in AutoDoc

**Markergenerierung** = add the M8 named destinations to the existing M9.

**Xref generierung** = Interpret the Named Destination file und get the page number from the NDX file to say "see qTlvFindObject at API.PDF: page 14".

Wenn wir die **BKX** Dateien nehmen, dann haben wir sogar den **Full Title** ! Aber vielleicht ist es besser, wenn man das Kürzel der ND nimmt und einfach Chapter 9.3.1 daraus macht.

**HS\_25Mar06:** Hadere immer noch damit, daß in der FrameMaker Datei letztlich nicht mehr das Target aufgelöst wird. Dort ist nur "FileInfo" und "Hash + Text" notwendig, die FileNo generierung findet erst im Distiller statt.

Hätte ich irgendwo die Information, WO eine File für einen bestimmten PreFix landet, dann könnte ich auflösen. Implizit ist dieses WO nur für das Target PDF aus der NDX File ersichtlich.

Abhilfe ... ich könnte den Directory Tree scannen auf die Existence der target file. Dann habe ich im zweiten Anlauf mit "API\<find it>" einen Hit auf die Target file und kann dann den M9 Marker automatisch setzen. Daher dient der Prefix im XREF dazu, die Anchor Directory zu finden.

-----

Konstruktion: Brauche eine Routine, die mir **TargetPathName(Prefix, FileName)** liefert. Dieses Ergebnis wird dann automatisch als Cross Referece eingesetzt. Das ersetzt quasi genau die Liste, die aus SECCOSREF.FM gefeederd wird, mit dem Unterschied, daß alle Marker aufgenommen werden. Der Anchor muss mit Prefix abgelegt werden, (so, wie es auch in der SECCOSREF.FM passieren müsste).

---

## 9.4 Final Design of AutoXref and AutoMarker

Processing AUTODOC generates an XRF file in the <refdir>\XRF\<sourcename.XRF>.

That file contains entries like

```
qTlvFindObject,sTlvLib.fm,"44441: Function: qTlvFindObject"
```

The first entry is the Token, the second entry the source file name and the third field is the cross reference text as being required by FrameMaker.

When AUTODOC parses through the NDX files, it learns the available host paths (like API, IMP etc).

For the auto cross reference, the system tries to resolve the token from the AxrList and the path from the information in the PreFixList (stores host directory) and by searching the host directory for the appropriate source file.

From this the information is available for xref.

The environment variable "PreFix" (case sensitive !) is evaluated in order to provide a run-time prefix. If an entry is made with "**env**\_qTlvFindObject" then the **env** triggers to search the actual prefix in the "PreFix" environment variable.



# 10 Call Graph Generator

## 10.1 Introduction

The FCT call graph generator uses ASCII source file input to generate VISIO files that display the calling tree dependencies of functions of an entire project.

Any called or calling functions may be written in any file that is part of the project. The only information needed to be provided to the call graph tool is a list of files that contains every source file that may hold a function definition (See “Function definition” on page 10-185.).

Currently the FCT program is a command line program, the parameters being entered in the command line when the program is invoked.

The output of the FCT program is a VISIO file, that contains the call graph. The produced call graph follows the rules of the SmartCard module layering such that it produces the colors that comply with project specific definitions.

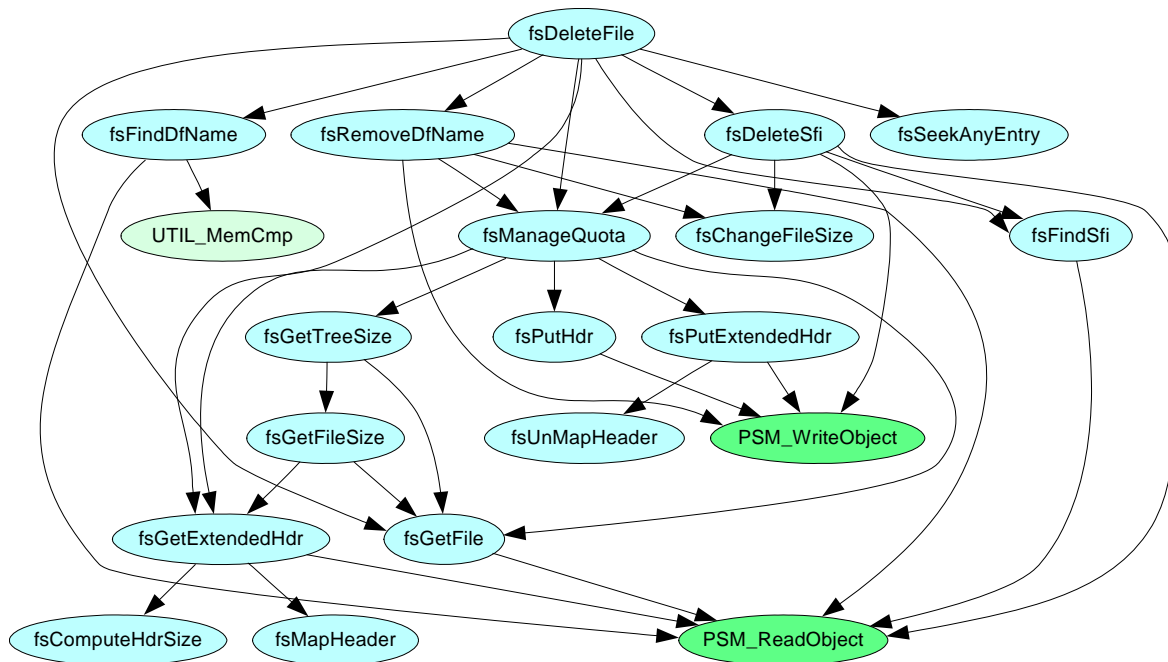


Figure 10-1. Example of a call graph

Figure 10-1 shows an example of a generated call graph. In order to know about the existence and relation of a called function, every project file was visited by the FCT program.

The colors in the graph reflects the modularity layer, this will be discussed in further detail in 10.2.3 “Layer processing” on page 10-186.

## 10.2 How to generate a call graph

A call graph is generated by the FCT.EXE.

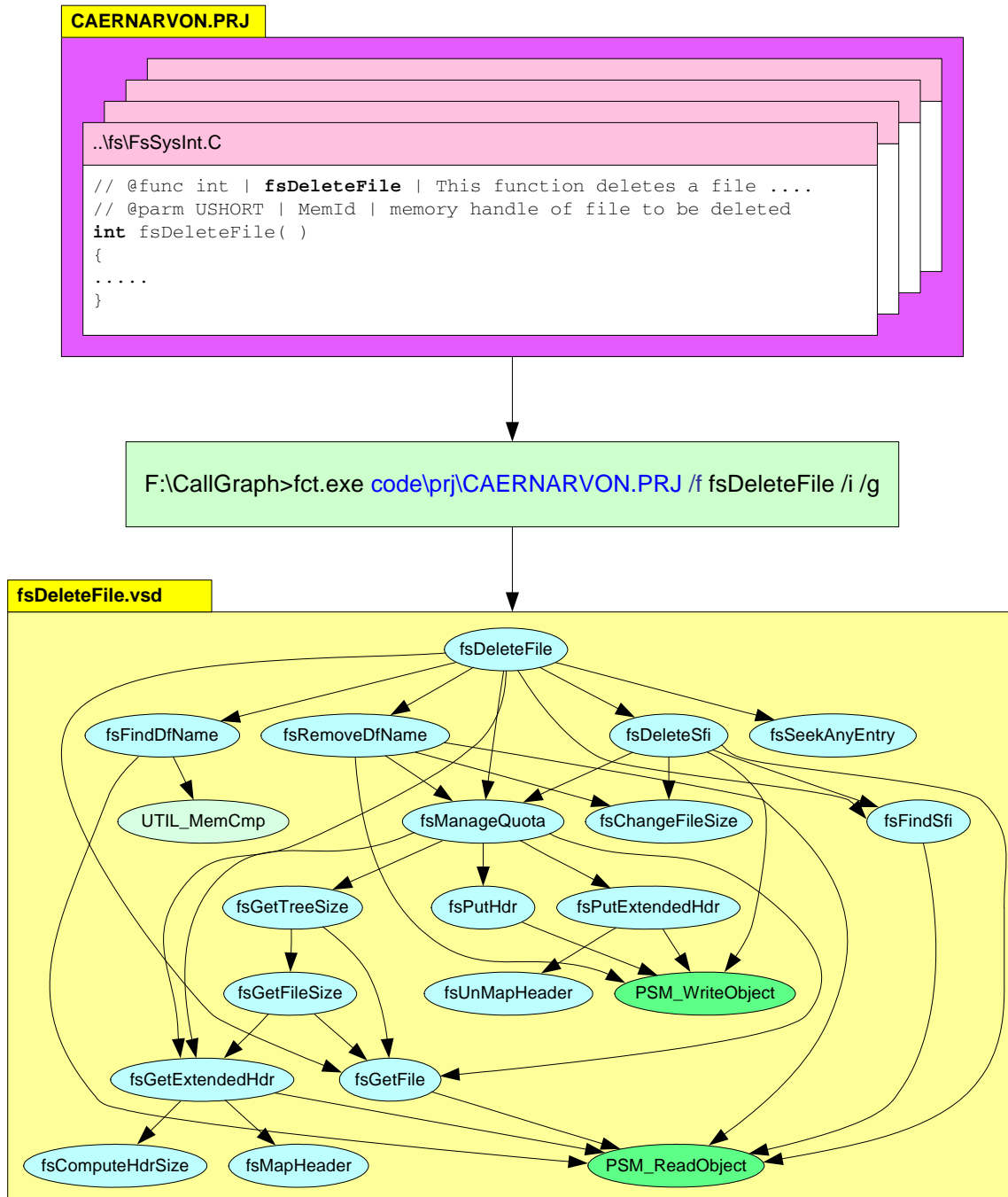


Figure 10-2. Generate call graph from source

The basic input information to the FCT.EXE is the project file and the function name of the function to be evaluated.

## 10.2.1 Command line options

Several command line options are available. These options are displayed if the FCT.EXE is called without any input parameter

```
FCT.EXE Version 2.1 (C) IBM Corp. 2004

This program generates a file that contains the calling tree information for
Caernarvon modules. The information is derived from the Raisonance/CodeWright
project file (.PRJ/.PJT) or the Raisonance cross reference file (.XRF)

Author: Helmut Scherzer, Dept. 7828, +49 7071 42754
        scherzer@de.ibm.com

Invoke with: D:\>fct CAERNARVON.<ext> [/g] [/i] [/n<maxdepth=0>] [/f<funcname>]

The extension .PRJ is recommended and uses Raisonance project files
The extension .PJT may be used for CodeWright project files
The extension .XRF may be used for Raisonance cross reference files, however
using .XRF does not find the local function declarations

Options:
=====
/g required once after a fresh compile)
/nx x is the max. nesting level, x=0 develops until modularity level change
/i ignore warnings, don't stop processing to wait for SPACE/RET
/f specifies a particular function to be picked for evaluation

Example: D:\>fct prj\caernarvon.prj /f fsDeleteFile /i /g
visits all source files in CAERNARVON.PRJ but evaluates only fsDeleteFile

No input defined - Program stopped

F:\Comps\projects\fct>
```

Figure 10-3. Empty invocation of FCT.EXE

- |                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>/g</b>                                                                                                                                                                                                                                                                                                                                                                               | generate function list. In order to resolve a call graph, the call graph generator needs to maintain a list of every function that is declared in the project. The generation of this function list takes some time, however it is only required once after a project compile. |
| Therefore the option /g allows a higher performance for the generation of a call graph.                                                                                                                                                                                                                                                                                                 |                                                                                                                                                                                                                                                                                |
| <b>/n &lt;x=0&gt;</b>                                                                                                                                                                                                                                                                                                                                                                   | nesting level. This parameter is typically not used as its default value 0 is already the recommended mode.                                                                                                                                                                    |
| The nesting level determines the calling depth of a call graph. x=1 shows only the functions, that are immediately called by the investigated function (/f). x=2 shows also the function that are called by those function being called by the investigated function (/f). As higher the nesting level is, the more complex the call graph looks like.                                  |                                                                                                                                                                                                                                                                                |
| x=0 is the default value. This has a special meaning. It will develop the call graph up to the next modularity level change. The nesting level is dynamic, nested calls are stopped if the last recent called function has a different modularity layer than the calling function. That last function will be included in the call graph, however, the functions being called will not. |                                                                                                                                                                                                                                                                                |
| <b>/i</b>                                                                                                                                                                                                                                                                                                                                                                               | Ignore warnings. This parameter avoids, that the FCT.EXE stops on warnings. Unless this parameter is given, on warnings the FCT.EXE suspends execution and waits for user input. A SPACE key continues processing, while a RET key will abort the program.                     |

When `/i` is set, then those warnings do not stop the execution flow.

In any case, the warning messages are recorded in ERR.LOG file.

**`/f <name>`** function selection. If this parameter is present, only the function **`<name>`** is evaluated. **`<name>`** is case sensitive.

If a non-existing name is presented, no call graph (VISIO file) will be generated.

If the parameter **`/f`** is not presented, then **all** functions of the project will be evaluated. This generates as many VISIO files as functions are in the project. Typically this is not recommended.



## 10.2.2 Function definition

Besides the graphical representation, the challenge for the call graph generation is to find the called function within a function. In the first place the function names need to be ‘learned’ by the call graph generator.

Several methods have been evaluated for this purpose, however, only one method has been found versatile and rich enough to finally allow a proper evaluation. The function names have to be ‘declared’ in function headers.

```
////////////////////////////////////  
// @func Q_RETURN | fsDeleteFile | Delete a file specified by <v>MemId<ef>  
//  
// @parm USHORT | AvQuota | the actual parameter is not required, however,  
//                      this parameter must exist to allow this function to be called  
//                      by <xref fsVisitChildren> for the removal of a directory tree.  
//  
//                      To save variables, therefore this parameter is actually used  
//                      in this function to store the value of <v>AvQuota<ef>  
//  
// @parm USHORT_P | MemId | <sv MemId> of the file to be deleted  
// @desc deletes <v>MemId<ef> of any file type and sets the Flags in the  
//       <v>FileHandles<ef>. This function does not check the access conditions, neither  
//       does it check the file type. Therefore the caller must be aware, that deleting  
//       a DF could break the parent chain of children under this DF. Care has to be  
//       taken for this situation by the caller. If a DF is to be deleted, then the  
//       <xref fsDeleteSubTree> should be taken instead in order to properly delete an  
//       entire tree.  
//  
Q_RETURN fsDeleteFile(USHORT    MemId,  
                      IFINFO_P  HdrInfo,  
                      USHORT     AvQuota,  
                      UCHAR_P    ArgV[]){  
  
    USHORT SfiOffset;  
    USHORT LastOffset;  
    ...  
    <code ....>  
}
```

Figure 10-4. Example of a function declaration

The call graph generator finds a function name by parsing the **@func** tags in the comments of function headers. In the above example, only the **blue** part is required for the correct call graph processing. However, it is recommended anyway to follow the syntax of the AUTODOC specification in order to allow further machine readable evaluation.

### Rationale

One could easily argue, that scanning function names in comments is not a good method instead of parsing the actual source. However, this method has proven to be quite powerful. It allows sources of most different nature to be scanned without knowing the actual language that they are written in. For instance, a test sequence could use the same syntax, given the call graph generator can evaluate, ‘what a comment is’, it can actually develop a call graph in an unknown programming language. This is the reason, why the call graph generator can be applied to assembler source code as well.

Regarding assembler code another aspect becomes obvious. In assembler code, a programmer might have hundreds of very small ‘calls’ that he would intentionally not want to be part of a call graph in order to avoid unnecessary complexity. If these functions are not declared with a **@func** tag (but with an **@subfunc** tag instead) then they are not recognized by the call graph generator.

Also parsing assembler source for function calls is much more complicated and can lead to undesirable results. Using the comment declarations allows a much clearer indication of ‘what is a function’.

A third argument relates to 'C' code again. A function can also be coded as a macro. As macro definitions may become quite complex, the call graph generator might have to become kind of a C-preprocessor in order to 'understand' the program syntax. This however, would certainly limit the versatility of the call graph generator.

Therefore it was decided to use the **@func** declarations in order to evaluate the call graph. Tests with several other schemes have shown that this is a favorite approach.

The disadvantage of this method is, that it might happen, that there are misspellings in the **@func** tags with respect to the actual coded function name. As far as these misspellings are only the capitalization of letters in the function's name, the call graph generator produces a warning.

### 10.2.3 Layer processing

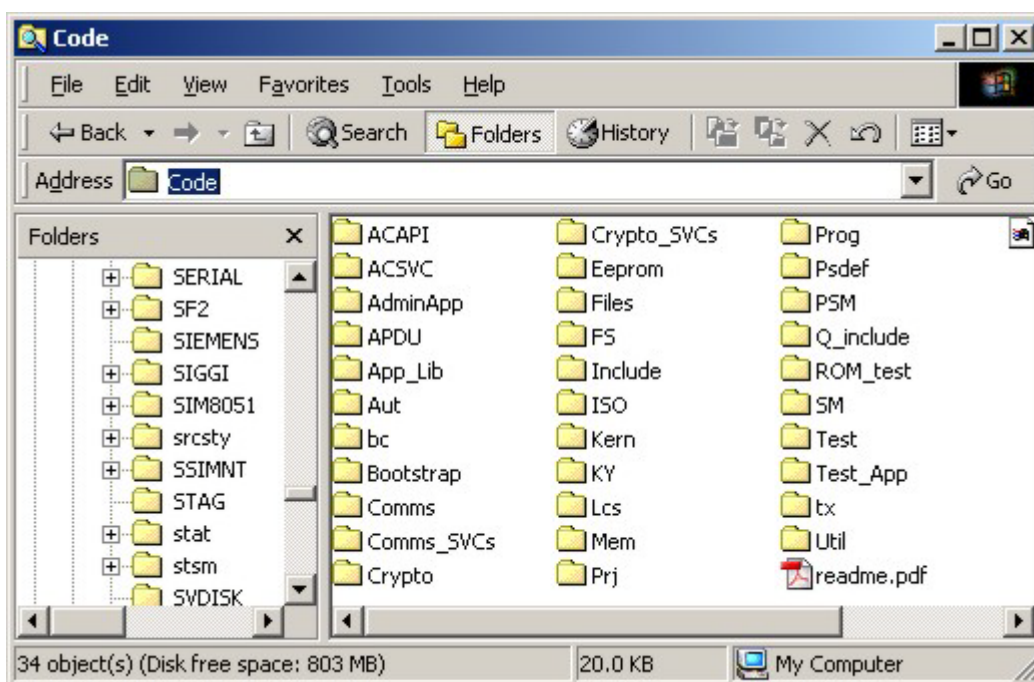


Figure 10-5. Project directories of the SmartCard project.

The call graph generator determines the correct layer from the *parent directory name* of a source file. The assignment of layers to directories is hard coded, a change requires a recompile.

**Author's Note:** On demand, this may be subject to change in later versions

The call graph generator will produce the same colors in the 'function bubbles' as used in Figure 10-2. If the command line parameter **/n0** was used (or just omitted from the command line as **/n0** is the default value anyway) then the call graph tree will stop further branch development.

### 10.2.4 Using MACROS.REF

The FCT.EXE uses a project file (either a Raisonance project (.PRJ) file, or a more popular CodeWright project file (.PJT) in order to determine the set of files that belong to a project. It may happen, however, that some functions are declared in files, that cannot easily be found in a project file. This applies for macro definitions that are typically coded in include files (.H). Those include files are sometimes not found in a project file, in particular they are not easily retrievable in a Raisonance project file.

Again, parsing the actual source for retrieving #include and nested #include statements was not considered best practice. Instead a file **MACROS.REF** is placed in the same directory as the project file.

MACROS.REF contains a list of source files to be added to the list of source files, found in the project file. For a CodeWright project file, this is not necessary, but can still be of value regarding to the following aspect.

In particular when included .H files might be collected in a common directory e.g. **..\\include**. it is not possible, to determine the layer of a macro/function coded in such a .H file from the **..\\include** directory, in particular it might be the case, that functions belonging to different layers, may be macro-defined in the same .H file.

The additional file MACROS.REF allows a specific overriding specification of particular functions to source files and layers. Any layer specification that is an exception from the global rule, that parent directories specify layer, can be coded in MACROS.REF. Those information will override the information that was found from parsing the entire project tree. It will also add functions, that are not listed in the project file for whatever reason.

It is also possible to define just additional files, without a particular function specification. Then those files will be parsed too to find **@func** tags of functions to be added to the list of available functions.

```
; This file defines the macros
; #1 = Function Name,          #2 = SourceFile      #3 = Directory that determines the layer
;
PSM_ReadObject,                bc\\DEFFUNC.C,      ..\\base\\PSM
PSM_WriteObject,               bc\\DEFFUNC.C,      ..\\base\\PSM
UTIL_OSPtrArgMakeFromObject,   bc\\DEFFUNC.C,      UTIL
UTIL_OSPtrNamesObj,           include\\util.h,    UTIL
UTIL_OSPtrArgBumpOffset,       include\\util.h,    UTIL

; an empty entry in #2 only adds a file to the list of functions to be
; searched for @func tags
                                ,      include\\PSM.H,
```

Figure 10-6. Example of MACROS.REF

In the above example, several functions are assigned to override possible entries from the project scan. The layer is determined by a directory name, that associates the desired layer.

## 10.2.5 Using LAYERS.REF

LAYERS.REF is a file the specifies the layers associated with some project directories. This file is mandatory, if the /n=0 option “Stop at layer change” is selected or if colorerd bubbles are wanted.

The directory notation is relative to the **..\\<projectdir>** path, i.e. if ICC.PJT is in **...\\base\\sim\\pjt**, then the directory notation will be from **...\\base\\sim**.

A typical content of the LAYERS.REF could be.

```
..\\App\\Applib,                0
..\\App\\BaseCmd,              0
..\\App\\Cpa,                  1
..\\App\\Gk,                   1
..\\App\\Include,              2
..\\App\\Tan,                  2
..\\base\\APDU,                 3
..\\base\\bootstrap,           3
```

Figure 10-7. Example of LAYERS.REF

where the first field represents the path, relative to `..\<projectpath>` and the second field represents a 0-base number that belongs to a layer. Each layer has a pre-assigned color in the FCT.EXE.

## 10.2.6 Generated output files

When running FCT.EXE with the option /g, several files are generated

**<projectdir>\FUNC.TXT**

contains a list of all functions followed by a 0-based line index into `..\bc\SRC.TXT`.

**<projectdir>\SRC.TXT**

contains a list of all source files including the full path definition

**<projectdir>\REF.TXT**

contains all functions and their immediately called functions. The called functions are indented. This is the basic information to generate the call graph.

---

## 10.3 Installation package

The following files are required to properly install the FCT call graph generator.

**FCT.EXE**

The call graph generating program

**DOT.EXE**

Public domain open source software contained in the GraphWiz package. See <http://www.research.att.com/sw/tools/graphviz/> for further info

**FT.DLL**

**JPEG.DLL**

**PNG.DLL**

**Z.DLL**

DLL's required by **DOT.EXE**.

**BOOKMSTR.VST**

Bookmaster template that is used by FCT.EXE in order to select the layer colors, drawing layout and stencils

**MACROS.REF**

Text file to override or add function definitions as described in [10.2.4 "Using MACROS.REF"](#).

All files but MACROS.REF should be installed in the same directory. The DLL files should be placed in a directory that is part of the PATH= environment variable or the installation path of FCT.EXE is to be added to the PATH= environment variable.

---

# 11 Glossary and Abbreviations

---

## 11.1 Abbreviations

<b>AHW</b>	Hardware Architecture
<b>ASW</b>	Software Architecture
<b>ASY</b>	System Architecture
<b>CRQ</b>	Change Request
<b>CRS</b>	Customer Requirements Specification
<b>CRT</b>	Certification documents
<b>DMA</b>	Direct Memory Access
<b>ECC</b>	European Citizen Card
<b>EMC</b>	Electromagnetic Compatibility
<b>IMP</b>	Implementation documentation
<b>REQ</b>	Requirements documentation
<b>SBD</b>	Storyboard
<b>STARS</b>	→ <a href="#">2.3.3 “Stakeholder Requirement Specification (STARS)”</a> on page 2-17
<b>SysRS</b>	→ <a href="#">2.3.4 “System Requirement Specification (SysRS)”</a> on page 2-17
<b>TOE</b>	Target of evaluation
<b>TST</b>	Test architecture
<b>VISIO</b>	MicroSoft Visio Drawing Tool
<b>UML</b>	Unified Modeling Language
<b>URS</b>	User Requirements Specification

## A

<b>Application</b>	Specific use of functional interfaces provided by a generic system
<b>ARC42</b>	A template for architecture documents from Dr. Peter Hruschka und Dr. Gernot Starke. The current version is 3.2 (January 2008). It is available under the Creative Common Licence, which allows free usage even in commercial environment. For further information, see <a href="http://www.arc42.de/template/template.html">http://www.arc42.de/template/template.html</a> .
<b>Architecture</b>	Description of composition and interaction of functional units (components) in a specified context.
<b>AURS</b> (deprecated)	<p>Answer to User Requirements Specification (→ <a href="#">URS</a>)</p> <p>The AURS is written by the development group as a specification of what and how the requirements in the <a href="#">URS</a> can be met. The AURS typically contains raw design proposals and corresponding cost estimates in order to allow a discussion between ordering dept and development on the most appropriate realization. This process may iterate the <a href="#">URS</a> as well as the AURS to finally match the requirements and their realization.</p> <p>G&amp;D does no more support the concept of URS/AURS but replaced it by the more efficient <a href="#">STARS</a>.</p>
<b>Authentication</b>	<p>Authentication is the act of establishing or confirming something (or someone) as authentic, that is, that claims made by or about the thing are true. This might involve confirming the identity of a person, the origins of an artifact, or assuring that a computer program is a trusted one.</p> <p><a href="http://en.wikipedia.org/wiki/Authentication">http://en.wikipedia.org/wiki/Authentication</a></p>

## C

<b>CA</b>	<p>In cryptography, a certificate authority or certification authority (CA) is an entity which issues digital certificates for use by other parties. It is an example of a trusted third party. CAs are characteristic of many public key infrastructure (PKI) schemes. There are many commercial CAs that charge for their services. There are also several providers issuing digital certificates to the public at no cost. Institutions and governments may have their own CAs.</p> <p><a href="http://en.wikipedia.org/wiki/Certification_authority">http://en.wikipedia.org/wiki/Certification_authority</a></p>
<b>CC EALn</b>	<p>The Common Criteria for Information Technology Security Evaluation (abbreviated as Common Criteria or CC) is an international standard (IEC 15408) for computer security.</p> <p>Common Criteria is based upon a framework in which computer system users can specify their security requirements, vendors can then implement and/or make claims about the security attributes of their products, and testing laboratories can evaluate the products to determine if they actually meet the claims. In other words, Common Criteria provides assurance that the process of specification, implementation and evaluation of a computer security product has been conducted in a rigorous and standard manner.</p> <p><a href="http://en.wikipedia.org/wiki/Common_criteria">http://en.wikipedia.org/wiki/Common_criteria</a></p>
<b>Component</b>	Functional unit with a specified interface

<b>CSV</b>	<p>Comma separated value. A method to separate fields in an ASCII file. Every line of the ASCII file represents the row of a table or a data record. The fields are separated by commas.</p> <p><b>Example:</b> VIP,19891022,"Scherzer, Anne-Christine","Anki"</p> <p>contains the 4 fields: [VIP] [19891022] [Scherzer, Anne-Christine] ["Anki"]</p> <p>To express a comma within the field, the entire text of the field is put in quotes ". Hence a comma within the quotes will not be interpreted as field separator.</p> <p>To express a quote "-character, <i>double quotes</i> will be used</p>
<b>D</b>	
<b>DDE</b>	<p>Dynamic Data Exchange : A mechanism to allow inter-application-communication. If an application provides a DDE server functionality, another application may address this server and exchange messages.</p> <p>The Adobe Reader product, for instance, allows some functions to be executed via DDE e.g. "Open Document at page .." or Open Document at Named Destination".</p>
<b>DOORS</b>	DOORS requirement management tool
<b>DÜESS</b>	a basic format description of a personalization order and the official verb for the entire idea of a personalization description.
<b>E</b>	
<b>EMF</b>	→ WMF
<b>EPS</b>	<p>Encapsulated PostScript, or EPS, is a DSC-conforming PostScript document with additional restrictions intended to make EPS files usable as a graphics file format.</p> <p>In other words, EPS files are more-or-less self-contained, reasonably predictable PostScript documents that describe an image or drawing, that can be placed within another PostScript document.</p> <p><a href="http://en.wikipedia.org/wiki/Encapsulated_PostScript">http://en.wikipedia.org/wiki/Encapsulated_PostScript</a></p>
<b>F</b>	
<b>Feature</b>	Behavior of a basic system that fulfills a specified use case
<b>Fine architecture</b>	Architecture that provides a complete and sufficient description of components, mechanisms and interfaces such that an unmistakable implementation of the described interfaces and function is possible.
<b>FIPS</b>	<p>Federal Information Processing Standards (FIPS) are publicly announced standards developed by the United States Federal government for use by all non-military government agencies and by government contractors.</p> <p>Many FIPS standards are modified versions of standards used in the wider community (ANSI, IEEE, ISO, etc.)</p> <p><a href="http://en.wikipedia.org/wiki/Federal_Information_Processing_Standard">http://en.wikipedia.org/wiki/Federal_Information_Processing_Standard</a></p>
<b>Framework</b>	Sufficient specification of mandatory interfaces and rules within which a variety of implementations is possible and allowed.
<b>O</b>	

**OLE** Object Linking and Embedding (OLE) is a technology that allows embedding and linking to documents and other objects developed by Microsoft. For developers, it brought OLE custom controls (OCX), a way to develop and use custom user interface elements.

For example, a *desktop publishing system* might send some text to a *word processor* or a picture to a *bitmap editor* using OLE. The main benefit of using OLE is to display visualizations of data from other programs that the host program is *not normally able to generate itself* (e.g. a pie-chart in a text document), as well as to create a master file.

[http://en.wikipedia.org/wiki/Object\\_Linking\\_and\\_Embedding](http://en.wikipedia.org/wiki/Object_Linking_and_Embedding)

**Operating System** Software providing at least interfaces and associated functionality in order to manage the hardware variants of possible computing systems

## P

**Platform** System providing a basic behavior and a set of interfaces and associated functions in order to implement additional use cases or products by using these provisions.

**PNG** Portable Network Graphics (PNG) is a bitmapped image format that employs lossless data compression. PNG was created to improve upon and replace GIF (Graphics Interchange Format) as an image-file format not requiring a patent license.

PNG supports palette-based (palettes of 24-bit RGB colors), greyscale or RGB images. PNG was designed for transferring images on the Internet, not professional graphics, and so does not support other color spaces (such as CMYK). The major advantage of PNG graphics is that they support transparency.

[http://en.wikipedia.org/wiki/Portable\\_Network\\_Graphics](http://en.wikipedia.org/wiki/Portable_Network_Graphics)

**Product** A physical thing or a service which is subject to trade and legal evidence in commercial context

## R

**Raw architecture** Architecture that only focuses on a minimum of components, functions and interfaces necessary in order to understand the basic function of the context or target system

## S

## T

**Toolbox** Set of components with sufficiently specified interfaces to allow the combination of these components to form a functional unit whose purpose is not predictable by the components itself



## U

### URL

In computing, a Uniform Resource Locator (URL) is a Uniform Resource Identifier (URI) which also specifies where the identified resource is available and the protocol for retrieving it.

In popular usage and in many technical documents it is often confused as a synonym for *uniform resource identifier*.

Every URL begins with the scheme name that defines its namespace, purpose, and the syntax of the remaining part of the URL. Most Web-enabled programs will try to dereference a URL according to the semantics of its scheme and a context-vbn. For example, a Web browser will usually dereference the URL `http://example.org/` by performing an HTTP request to the host *example.org*, at the default HTTP port (port 80).

Dereferencing the URL `mailto:bob@example.com` will usually start an e-mail composer with the address *bob@example.com* in the *To field*.

*example.com* is a domain name; an IP address or other network address might be used instead. In addition, URLs that specify `https` as a scheme (such as `https://example.com/`) normally denote a *secure* website.

<http://en.wikipedia.org/wiki/URL>

### URS

(deprecated)

User Requirement Specification, a document which belongs to project documentation specifying the raw function and non-functional requirements of the product.

The URS shall be descriptive enough in order to allow a cost/PM estimation for the development/testing with focus on a defined exit level (prototype level, product level etc.).

G&D does no more support the concept of URS/AURS but replaced it by the more efficient STARS.

## W

### WMF

Windows Metafile (WMF) is a graphics file format on Microsoft Windows systems, originally designed in the early 1990s. Windows Metafiles are intended to be portable between applications and may contain both vector and bitmap components. In contrast to raster formats such as JPEG and GIF which are used to store bitmap graphics such as photographs, scans and graphics, Windows Metafiles generally are used to store line-art, illustrations and content created in drawing or presentation applications. Most Windowsclipart is in the WMF format.

[http://en.wikipedia.org/wiki/Enhanced\\_Metafile](http://en.wikipedia.org/wiki/Enhanced_Metafile)



---

# Appendix A. FrameMaker Character Sets

This appendix lists the character sets used for FrameMaker documents using Western fonts when running under Windows, and shows how to type each character in the set.

**Note:** This appendix is derived from a FrameMaker “Online Manual” issued with FrameMaker. This information is included here for convenience of the user.

## A.0.0.1 About character sets

FrameMaker uses three kinds of character sets.

- Dingbat character set - for the Zapf Dingbats font
- Symbol character set - for the Symbol font
- Standard character set - for all other fonts

These three character sets include not only the keys on the keyboard, but also many special characters such as mathematical symbols, accented letters, and a variety of dingbats such as arrows and stars.

**Note:** If the desired character is in the Symbol or Zapf Dingbats character set and that is not the current font, it is necessary to change the font before typing the character.

The Windows character set is based on the ANSI character set, and includes some additional characters not in the ANSI set. On platforms other than Windows, FrameMaker uses a character set based on Adobe PostScript instead of ANSI. A few of the characters in the PostScript set are not available in the ANSI set:

fl and fi ligatures, dotlessi, breve, dotaccent, and ogonek

If an open document that was created on another platform in the Windows version of a FrameMaker product, an underline character appears in place of any characters not available in the ANSI set. FrameMaker preserves the code of the original characters; if the document is subsequently opened on the other platform, the correct characters reappear.

Some character values are reserved for future use. Although several of these values cause characters to appear in a document window, they can cause different characters or no character to appear when printed. Also, they may not produce the same characters on different platforms. If it is intended to transfer files between platforms, the characters marked *Reserved* in the following tables should not be used.

The code values in the following tables appear in hexadecimal notation in columns labeled “Hex code”. The hexadecimal code shown for each character is the code that is used to represent the character internally in FrameMaker. If you’re using Maker Markup Language (MML) or another program that creates files in Maker Interchange Format (MIF), you may need to refer to the codes from time to time.

The characters at the beginning of the table, with hexadecimal codes below 0x20, are called control codes. Rather than specifying characters to be printed, these characters affect how surrounding text is formatted. Some of these characters are visible in a document window if text symbols are showing.

The instructions for typing quotation marks and apostrophes assume that Smart Quotes is off. For information on Smart Quotes see the *FrameMaker User Guide*.

## A.0.0.2 Using key sequences

Many characters are generated by a key sequence. This key sequence often uses the Control or Esc key.

This manual uses the following conventions for key sequences:

*Table A-1. FrameMaker key sequences for special characters*

Example	Describes
Control+q	Holding down Control while pressing the lowercase letter <i>q</i>
Control+q Shift+a	Holding down Control while pressing the letter <i>q</i> , then releasing both keys, and then holding down Shift while pressing the letter <i>a</i>
Esc ~ Shift+a	Pressing and releasing Esc, then pressing and releasing ~ (tilde), then holding down Shift while pressing the letter <i>a</i>

Special characters may also be typed into a document by using its ANSI number:

1. Press Num Lock to make the numeric keypad active.
2. Hold down the Alt key while typing the ANSI number (including the leading zero) using the keys on the numeric keypad.

For example, to enter the “questiondown” character (¿) using its ANSI number, hold down the Alt key while typing 0191 from the keypad, and then release the Alt key. Be sure to include the leading zero.

In the following tables, where there exist two different keystroke sequences to type a character, the sequences are separated by a comma.

## A.0.1 The Windows character sets

The following table shows all the characters available in the Windows version of FrameMaker products. It starts with the special hyphens, spaces, and returns a user can enter, and then lists the rest of the characters in their ANSI order.

### A.0.1.1 Special Characters - Hyphens, Spaces, Etc.

*Table A-2. Special hyphens, spaces, returns, and un-displayed characters (Part 1 of 2)*

FrameMaker Hex code	Key or key sequence	Standard character set: graphic and name	Symbol set: graphic and name	Dingbats: graphic
0x04	Esc hyphen Shift+d, Control+hyphen	discretionary hyphen	discretionary hyphen	
0x05	Esc n s	suppress hyphenation	suppress hyphenation	
0x15	Esc hyphen h	nonbreaking hyphen	nonbreaking hyphen	
0x08	Tab	tab	tab	
0x09	Shift+Return	forced return	forced return	
0x0a	Return	end of paragraph	end of paragraph	
0x10	Esc space 1 (one)	numeric space	numeric space	
0x11	Esc space h, Control+space	nonbreaking space	nonbreaking space	
0x12	Esc space t	thin space	thin space	
0x13	Esc space n, Alt+Control+space	en space	en space	

Table A-2. Special hyphens, spaces, returns, and un-displayed characters (Part 2 of 2)

FrameMaker Hex code	Key or key sequence	Standard character set: graphic and name	Symbol set: graphic and name	Dingbats: graphic
0x14	Esc space m, Control+Shift+space	em space	em space	
0x27	Control+'	' quotesingle	ə such that	☹
0x60	Control+`	` grave	— radicalex	☼
0xda	Control+q Shift+z	• fraction		
0xde	Control+q ^	• Un-displayed <sup>1</sup>		
0xdf	Control+q _	• Un-displayed <sup>1</sup>		
0xf5	Control+q u	? Un-displayed <sup>1</sup>		
0xf9	Control+q y	˘ Un-displayed <sup>1</sup>		
0xfa	Control+q z	Ž Un-displayed <sup>1</sup>		
0xfe	Control+q ~	ž Un-displayed <sup>1</sup>		
0xfd	Control+q }	” hungarumlaut		

<sup>1</sup> the FrameMaker manual claims this character is “Un-displayed”. However, it would appear that a character is displayed.

### A.0.1.2 ANSI Character Set Codes 32<sub>10</sub>-127<sub>10</sub>

Table A-3. ANSI Character Set Codes 32<sub>10</sub>-127<sub>10</sub> (Part 1 of 4)

ANSI # / FrameMaker Hex code	Key or key sequence	Standard character set: graphic and name	Symbol set: graphic and name	Dingbats: graphic
32 / 0x20	space	space	space	
33 / 0x21	!	! exclam	! exclam	✂
34 / 0x22	“ (smart quotes off)	“ quotedbl	∀ universal	✂
35 / 0x23	#	# numbersign	# numbersign	✂
36 / 0x24	\$	\$ dollar	∃ existential	✂
37 / 0x25	%	% percent	% percent	☎
38 / 0x26	&	& ampersand	& ampersand	🕒
39 /	‘	‘ single quote <sup>1</sup>	ə such that	☹
40 / 0x28	(	( parenleft	( parenleft	✈
41 / 0x29	)	) parenright	) parenright	🇬🇧
42 / 0x2a	*	* asterisk	* asteriskmath	☞
43 / 0x2b	+	+ plus	+ plus	📧
44 / 0x2c	, (comma)	, comma	, comma	👉
45 / 0x2d	- (hyphen)	- hyphen	– minus	📎
46 / 0x2e	. (period or full stop)	. period or full stop	. period or full stop	📎

Table A-3. ANSI Character Set Codes 32<sub>10</sub>-127<sub>10</sub> (Part 2 of 4)


































ANSI # / FrameMaker Hex code	Key or key sequence	Standard character set: graphic and name	Symbol set: graphic and name	Dingbats: graphic
47 / 0x2f	/	/ slash	/ slash	
48 / 0x30	0	0 zero	0 zero	
49 / 0x31	1	1 one	1 one	
50 / 0x32	2	2 two	2 two	
51 / 0x33	3	3 three	3 three	
52 / 0x34	4	4 four	4 four	
53 / 0x35	5	5 five	5 five	
54 / 0x36	6	6 six	6 six	
55 / 0x37	7	7 seven	7 seven	
56 / 0x38	8	8 eight	8 eight	
57 / 0x39	9	9 nine	9 nine	
58 / 0x3a	:	: colon	: colon	
59 / 0x3b	;	; semicolon	; semicolon	
60 / 0x3c	<	< less	< less	
61 / 0x3d	=	= equal	= equal	
62 / 0x3e	>	> greater	> greater	
63 / 0x3f	?	? question	? question	
64 / 0x40	@	@ at	≡ congruent	
65 / 0x41	A	A A	A Alpha	
66 / 0x42	B	B B	B Beta	
67 / 0x43	C	C C	X Chi	
68 / 0x44	D	D D	Δ Delta	
69 / 0x45	E	E E	E Epsilon	
70 / 0x46	F	F F	Φ Phi	
71 / 0x47	G	G G	Γ Gamma	
72 / 0x48	H	H H	H Eta	
73 / 0x49	I	I I	I Iota	
74 / 0x4a	J	J J	ϑ theta1	
75 / 0x4b	K	K K	K Kappa	
76 / 0x4c	L	L L	Λ Lambda	
77 / 0x4d	M	M M	M Mu	
78 / 0x4e	N	N N	N Nu	
79 / 0x4f	O	O O	O Omicron	

Table A-3. ANSI Character Set Codes 32<sub>10</sub>-127<sub>10</sub> (Part 3 of 4)

ANSI # / FrameMaker Hex code	Key or key sequence	Standard character set: graphic and name	Symbol set: graphic and name	Dingbats: graphic
80 / 0x50	P	Π P	Π Pi	☆
81 / 0x51	Q	Θ Q	Θ Theta	✱
82 / 0x52	R	Ρ R	Ρ Rho	✱
83 / 0x53	S	Σ S	Σ Sigma	✱
84 / 0x54	T	Τ T	Τ Tau	✱
85 / 0x55	U	Υ U	Υ Upsilon	✱
86 / 0x56	V	ς V	ς sigma1	✱
87 / 0x57	W	Ω W	Ω Omega	✱
88 / 0x58	X	Ξ X	Ξ Xi	✱
89 / 0x59	Y	Ψ Y	Ψ Psi	✱
90 / 0x5a	Z	Ζ Z	Ζ Zeta	✱
91 / 0x5b	[	[ bracketleft	[ bracketleft	✱
92 / 0x5c	\	\ backslash	∴ therefore	✱
93 / 0x5d	]	] bracketright	] bracketright	✱
94 / 0x5e	^	^ asciicircum	⊥ perpendicular	✱
95 / 0x5f	_ (underline)	_ underscore	_ underscore	✱
96 / 0x60	‘ (grave) <sup>2</sup>	‘ grave <sup>2</sup>	⁻ radicalex <sup>2</sup>	✱ <sup>2</sup>
97 / 0x61	a	α a	α alpha	✱
98 / 0x62	b	β b	β beta	✱
99 / 0x63	c	γ c	γ chi	✱
100 / 0x64	d	δ d	δ delta	✱
101 / 0x65	e	ε e	ε epsilon	✱
102 / 0x66	f	φ f	φ phi	✱
103 / 0x67	g	γ g	γ gamma	✱
104 / 0x68	h	η h	η eta	✱
105 / 0x69	i	ι i	ι iota	✱
106 / 0x6a	j	φ j	φ phi1	✱
107 / 0x6b	k	κ k	κ kappa	✱
108 / 0x6c	l	λ l	λ lambda	●
109 / 0x6d	m	μ m	μ mu	○
110 / 0x6e	n	ν n	ν nu	■
111 / 0x6f	o	ο o	ο omicron	□
112 / 0x70	p	π p	π pi	□

Table A-3. ANSI Character Set Codes 32<sub>10</sub>-127<sub>10</sub> (Part 4 of 4)

ANSI # / FrameMaker Hex code	Key or key sequence	Standard character set: graphic and name	Symbol set: graphic and name	Dingbats: graphic
113 / 0x71	q	q q	θ theta	☐
114 / 0x72	r	r r	ρ rho	☐
115 / 0x73	s	s s	σ sigma	▲
116 / 0x74	t	t t	τ tau	▼
117 / 0x75	u	u u	υ upsilon	◆
118 / 0x76	v	v v	ω omega1	❖
119 / 0x77	w	w w	ω omega	◐
120 / 0x78	x	x x	ξ xi	
121 / 0x79	y	y y	ψ psi	!
122 / 0x7a	z	z z	z zeta	!
123 / 0x7b	{	{ braceleft	{ braceleft	‘
124 / 0x7c		bar	bar	’
125 / 0x7d	}	} braceright	} braceright	“
126 / 0x7e	~	~ asciitilde	~ similar	”
127				

1 note that this displays as a different character to the FrameMaker character code 0x27.

2 this appears to produce the same character as Control+singlequote

### A.0.1.3 ANSI Character Set Codes 128<sub>10</sub>-255<sub>10</sub>

The key stroke sequences required to input these character codes for the standard character set are different from the key strokes required for the symbol and dingbats character sets. Hence we display this range in two separate tables, one for the standard character set and one for the symbol/dingbat character set.

Table A-4. ANSI Character Codes 128<sub>10</sub>-255<sub>10</sub>, Standard Character Set (Part 1 of 3)

ANSI # / FrameMaker Hex code	Key or key sequence	Standard character set: graphic and name	ANSI # / FrameMaker Hex code	Key or key sequence	Standard character set: graphic and name
128			154 / 0xf0	Control+q p	š Reserved
129			155 / 0xdd	Control+q ]	› guilsinglright
130 / 0xe2	Control+q b	, quotesinglbase	156 / 0xcf	Control+q Shift+o	œ oe
131 / 0xc4	Control+q Shift+d	f florin	157		
132 / 0xe3	Control+q c	„ quotedblbase	158		
133 / 0xc9	Control+q Shift+i	... ellipsis	159 / 0xd9	Esc % Shift+y	ÿ Ydieresis
134 / 0xa0	Control+q space	† dagger	160		
135 / 0xe0	Control+q ‘	‡ daggerdbl	161 / 0xc1	Control+q Shift+a	¡ exclamdown
136 / 0xf6	Control+q v	^ circumflex	162 / 0xa2	Control+q “	¢ cent



Table A-4. ANSI Character Codes 128<sub>10</sub>-255<sub>10</sub>, Standard Character Set (Part 2 of 3)

ANSI # / FrameMaker Hex code	Key or key sequence	Standard character set: graphic and name	ANSI # / FrameMaker Hex code	Key or key sequence	Standard character set: graphic and name
137 / 0xe4	Control+q d	‰ perthousand	163 / 0xa3	Control+q #	£ sterling
138 / 0xb3	Control+q 3	Š Reserved	164 / 0xdb	Control+q [	¤ currency
139 / 0xdc	Control+q \	< guilsinglleft	165 / 0xb4	Control+q 4	¥ yen
140 / 0xce	Control+q Shift+n	Œ OE	166 / 0xad	Control+q hyphen	pipe
141			167 / 0xa4	Control+q \$	§ section
142			168 / 0xac	Control+q ,	¨ dieresis
143			169 / 0xa9	Control+q )	© copyrightserif
144			170 / 0xbb	Control+q ;	ª ordfeminine
145 / 0xd4	Control+q Shift+t, ‘	‘ quoteleft	171 / 0xc7	Control+q Shift+g	« guillemotleft
146 / 0xd5	Control+q Shift+u	’ quoteright	172 / 0xc2	Control+q Shift+b	¬ logicalnot
147 / 0xd2	Control+q Shift+r	“ quotedblleft	173 / 0x2d	- (hyphen)	- hyphen
148 / 0xd3	Control+q Shift+s	” quotedblright	174 / 0xa8	Control+q (	® registerserif
149 / 0xa5	Control +q %	• bullet	175 / 0xf8	Control+q x	˘ macron
150 / 0xd0	Control+q Shift+p	– endash	176 / 0xfb	Control+q {	° ring
151 / 0xd1	Control+q Shift+q	— emdash	177 / 0xb1	Control+q 1	± plusminus
152 / 0xf7	Control+q w	˜ tilde	178 / 0xb7	Control+q 7	² Reserved
153 / 0xaa	Control+q *	™ trademarkserif	179 / 0xb8	Control+q 8	³ Reserved
180 / 0xab	Control+q +	´ acute	210 / 0xf1	Esc ` Shift+o	ò Ograve
181 / 0xb5	Control+q 5	μ Reserved	211 / 0xee	Esc ‘ Shift+o	ó Oacute
182 / 0xa6	Control+q &)	¶ paragraph	212 / 0xef	Esc ^ Shift+o	ô Ocircumflex
183 / 0xe1	Control+q a	· period-centered	213 / 0xcd	Esc ~ Shift+o	õ Otilde
184 / 0xfc	Control+q ÿ	¸ cedilla	214 / 0x85	Esc % Shift+o	ö Odieresis
185 / 0xb6	Control+q 6	¹ Reserved	215 / 0xb0	Control+q zero	× Reserved
186 / 0xbc	Control+q <	º ordmasculine	216 / 0xaf	Control+q /	ø Oslash
187 / 0xc8	Control+q Shift+h	» guilemotright	217 / 0xf4	Esc ` Shift+u	ù Ugrave
188 / 0xb9	Control+q 9	¼ Reserved	218 / 0xf2	Esc ‘ Shift+u	ú Uacute
189 / 0xba	Control+q :	½ Reserved	219 / 0xf3	Esc ^ Shift+u	û Ucircumflex
190 / 0xbd	Control+q =	¾ Reserved	220 / 0x86	Esc % Shift+u	ü Udieresis
191 / 0xc0	Control+q @	¿ questiondown	221 / 0xc5	Control+q Shift+e	ý Reserved
192 / 0xcb	Esc ` Shift+a	À Agrave	222 / 0xd7	Control+q Shift+w	þ Reserved
193 / 0xe7	Esc ‘ Shift+a	Á Aacute	223 / 0xa7	Control+q ‘	ß germandbls
194 / 0xe5	Esc ^ Shift+a	Â Acircumflex	224 / 0x88	Esc ` a	à agrave
195 / 0xcc	Esc ~ Shift+a	Ã Atilde	225 / 0x87	Esc ‘ a	á aacute

Table A-4. ANSI Character Codes 128<sub>10</sub>-255<sub>10</sub>, Standard Character Set (Part 3 of 3)

ANSI # / FrameMaker Hex code	Key or key sequence	Standard character set: graphic and name	ANSI # / FrameMaker Hex code	Key or key sequence	Standard character set: graphic and name
196 / 0x80	Esc % Shift+a	Ä Adieresis	226 / 0x89	Esc ^ a	â acircumflex
197 / 0x81	Esc * Shift+a	Å Aring	227 / 0x8b	Esc ~ a	ã atilde
198 / 0xae	Control+q .	Æ AE	228 / 0x8a	Esc % a	ä adieresis
199 / 0x82	Esc comma Shift+c	Ç Ccedilla	229 / 0x8c	Esc * a	å aring
200 / 0xe9	Esc ` Shift+e	È Egrave	230 / 0xbe	Control+q >	æ ae
201 / 0x83	Esc ´ Shift+e	É Eacute	231 / 0x8d	Esc comma c	ç ccedilla
202 / 0xe6	Esc ^ Shift+e	Ê Ecircumflex	232 / 0x8f	Esc ` e	è egrave
203 / 0xe8	Esc % Shift+e	Ë Edieresis	233 / 0x8e	Esc ´ e	é eacute
204 / 0xed	Esc ` Shift+i	Ì Igrave	234 / 0x90	Esc ^ e	ê ecircumflex
205 / 0xea	Esc ´ Shift+i	Í Iacute	235 / 0x91	Esc % e	ë edieresis
206 / 0xeb	Esc ^ Shift+i	Î Icircumflex	236 / 0x93	Esc ` i	ì igrave
207 / 0xec	Esc % Shift+i	Ï Idieresis	237 / 0x92	Esc ´ i	í iacute
208 / 0xc3	Control+q Shift+c	Ð Reserved	238 / 0x94	Esc ^ i	î icircumflex
209 / 0x84	Esc ~ Shift+n	Ñ Ntilde	239 / 0x95	Esc % i	ï idieresis
240 / 0xb2	Control+q 2	ð Reserved	248 / 0xbf	Control+q ?	ø oslash
241 / 0x96	Esc ~ n	ñ ntilde	249 / 0x9d	Esc ` u	ù ugrave
242 / 0x98	Esc ` o	ò ograve	250 / 0x9c	Esc ´ u	ú uacute
243 / 0x97	Esc ´ o	ó oacute	251 / 0x9e	Esc ^ u	û ucircumflex
244 / 0x99	Esc ^ o	ô ocircumflex	252 / 0x9f	Esc % u	ü udieresis
245 / 0x9b	Esc ~ o	õ otilde	253 / 0xc6	Control+q Shift+f	ý Reserved
246 / 0x9c	Esc % o	ö odieresis	254 / 0xca	Control+q Shift+j	þ Reserved
247 / 0xd6	Control+q Shift+v	÷ Reserved	255 / 0xd8	Esc % y	ÿ ydieresis

Table A-5. ANSI Character Codes 128<sub>10</sub>-255<sub>10</sub>, Symbol/Dingbat Characters (Part 1 of 4)

ANSI # / FrameMaker Hex code	Key or key sequence	Symbol set: graphic and name	Dingbats: graphic
128 to 160	Reserved		
161 / 0xc1	Control+q !	Υ Upsilon1	☪
162 / 0xa2	Control+q “	’ minute	⋮
163 / 0xa3	Control+q #	≤ lessequal	⋮
164 / 0xdb	Control+q \$	/ fraction	♥

Table A-5. ANSI Character Codes 128<sub>10</sub>-255<sub>10</sub>, Symbol/Dingbat Characters (Part 2 of 4)

ANSI # / FrameMaker Hex code	Key or key sequence	Symbol set: graphic and name	Dingbats: graphic
165 / 0xa5	Control+q %	∞ infinity	♾
166 / 0xa6	Control+q &	f florin	ℳ
167 / 0xa7	Control+q ‘ <sup>2</sup>	♣ club	♣
168 / 0xa8	Control+q (	♦ diamond	♣
169 / 0xa9	Control+q )	♥ heart	♦
170 / 0xaa	Control+q *	♠ spade	♥
171 / 0xab	Control+q +	↔ arrowboth	♠
172 / 0xac	Control+q ,	← arrowleft	①
173 / 0xad	Control+q -	↑ arrowup	↗ <sup>1</sup>
174 / 0xae	Control+q .	→ arrowright	③
175 / 0xaf	Control+q /	↓ arrowdown	④
176 / 0xb0	Control+q 0	° degree	⑤ <sup>1</sup>
177 / 0xb1	Control+q 1	± plusminus	⑥ <sup>1</sup>
178 / 0xb2	Control+q 2	” second	⑦ <sup>1</sup>
179 / 0xb3	Control+q 3	≥ greaterequal	⑧ <sup>1</sup>
180 / 0xb4	Control+q 4	× multiply	⑨
181 / 0xb5	Control+q 5	∞ proportional	⑩ <sup>1</sup>
182 / 0xb6	Control+q 6	∂ partialdiff	① <sup>1</sup>
183 / 0xb7	Control+q 7	• bullet	② <sup>1</sup>
184 / 0xb8	Control+q 8	÷ divide	③ <sup>1</sup>
185 / 0xb9	Control+q 9	≠ notequal	④ <sup>1</sup>
186 / 0xba	Control+q :	≡ equivalence	⑤ <sup>1</sup>
187 / 0xbb	Control+q ;	≈ approxequal	⑥
188 / 0xbc	Control+q <	… ellipsis	⑦
189 / 0xbd	Control+q =	arrowvertex	⑧ <sup>1</sup>
190 / 0xbe	Control+q >	— arrowhorizex	⑨
191 / 0xbf	Control+q ?	↵ carriagereturn	⑩
192 / 0xc0	Control+q @	ℵ aleph	①
193 / 0xc1	Control+q Shift+a	℧ Ifraktur	②
194 / 0xc2	Control+q Shift+b	℞ Rfraktur	③

Table A-5. ANSI Character Codes 128<sub>10</sub>-255<sub>10</sub>, Symbol/Dingbat Characters (Part 3 of 4)

ANSI # / FrameMaker Hex code	Key or key sequence	Symbol set: graphic and name	Dingbats: graphic
195 / 0xc3	Control+q Shift+c	∅ weierstrass	④ <sup>1</sup>
196 / 0xc4	Control+q Shift+d	⊗ circlemultiply	⑤
197 / 0xc5	Control+q Shift+e	⊕ circleplus	⑥ <sup>1</sup>
198 / 0xc6	Control+q Shift+f	∅ emptyset	⑦ <sup>1</sup>
199 / 0xc7	Control+q Shift+g	∩ intersection	⑧
200 / 0xc8	Control+q Shift+h	∪ union	⑨
201 / 0xc9	Control+q Shift+i	⊃ propersuperset	⑩
202 / 0xca	Control+q Shift+j	⊇ reflexsuperset	❶ <sup>1</sup>
203 / 0xcb	Esc ` Shift+a <sup>2</sup>	⊈ notsubset	❷
204 / 0xcc	Esc ~ Shift+a	⊂ propersubset	❸
205 / 0xcd	Esc ~ Shift+o	⊆ relflexsubset	❹
206 / 0xce	Control+q Shift+n	∈ element	❺
207 / 0xcf	Control+q Shift+o	∉ notelement	❻
208 / 0xd0	Control+q Shift+p	∠ angle	❼
209 / 0xd1	Control+q Shift+q	∇ gradient	❽
210 / 0xd2	Control+q Shift+r	® registerserif	❾
211 / 0xd3	Control+q Shift+s	© copyrightserif	❿
212 / 0xd4	Control+q Shift+t	™ trademarkserif	➔
213 / 0xd5	Control+q Shift+u	∏ product	➔
214 / 0xd6	Control+q Shift+v	√ radical	↔ <sup>1</sup>
215 / 0xd7	Control+q Shift+w	· dotmath	↕ <sup>1</sup>
216 / 0xd8	Esc % y	¬ logicalnot	↘
217 / 0xd9	Esc % Shift+y	∧ logicaland	➔
218 / 0xda	Control+q Shift+z	∨ logicalor	↗
219 / 0xdb	Control+q [	↔ arrowdblboth	➔
220 / 0xdc	Control+q \	⇐ arrowdblleft	➔
221 / 0xdd	Control+q ]	⇑ arrowdblup	➔
222 / 0xde	Control+q ^	⇒ arrowdblright	➔
223 / 0xdf	Control+q _	⇓ arrowdbldown	➔
224 / 0xe0	Control+q ‘ <sup>2</sup>	◊ lozenge	➔
225 / 0xe1	Control+q a	⟨ angleleft	➔

Table A-5. ANSI Character Codes 128<sub>10</sub>-255<sub>10</sub>, Symbol/Dingbat Characters (Part 4 of 4)

ANSI # / FrameMaker Hex code	Key or key sequence	Symbol set: graphic and name	Dingbats: graphic
226 / 0xe2	Control+q b	® registersans	➤
227 / 0xe3	Control+q c	© copyrightsans	➤
228 / 0xe4	Control+q d	™ trademarksans	➤
229 / 0xe5	Esc ^ Shift+a	Σ summation	➤
230 / 0xe6	Esc ^ Shift+e	( parenlefttp	➤
231 / 0xe7	Esc ' Shift+a <sup>2</sup>	parenleftex	➤
232 / 0xe8	Esc % Shift+e	( parenleftbt	➤
233 / 0xe9	Esc ' Shift+e <sup>2</sup>	[ bracketlefttp	➤
234 / 0xea	Esc ' Shift+i <sup>2</sup>	bracketleftmid	➤
235 / 0xeb	Esc ^ Shift+i	[ brackleftbt	➤
236 / 0xec	Esc % Shift+i	{ bracelefttp	➤
237 / 0xed	Esc ' Shift+i <sup>2</sup>	{ braceleftmid	➤
238 / 0xee	Esc ' Shift+o <sup>2</sup>	[ braceleftbt	➤
239 / 0xef	Esc ^ Shift+o	braceex	➤
240 / 0xf0	Reserved		
241 / 0xf1	Esc ' Shift+o <sup>2</sup>	> angleright	➤
242 / 0xf2	Esc ' Shift+u <sup>2</sup>	∫ integral	➤
243 / 0xf3	Esc ^ Shift+u	∫ integraltp	➤
244 / 0xf4	Esc ' Shift+u <sup>2</sup>	integralex	➤
245 / 0xf5	Control+q u	∫ integralbt	➤
246 / 0xf6	Control+q v	) parenrighttp	➤
247 / 0xf7	Control+q w	parenrightex	➤
248 / 0xf8	Control+q x	) parenrightbt	➤
249 / 0xf9	Control+q y	] bracketrighttp	➤
250 / 0xfa	Control+q z	bracketrightex	➤
251 / 0xfb	Control+q {	] bracketrightbt	➤
252 / 0xfc	Control+q	} bracerighttp	➤
253 / 0xfd	Control+q }	} bracerightmid	➤
254 / 0xfe	Control+q ~	} bracerightbt	➤
255 / 0xff	Reserved		

<sup>1</sup> this character does not agree with that in the FrameMaker document. The FrameMaker document is believed to contain errors.

- 2 this key combination appears to not work at least some of the time - this may be operating system, hardware or context specific. Use Alt+keynumber as an alternative.

## **12 INDEX**

