

FDK PROGRAMMER'S GUIDE

ADOBE® FRAMEMAKER® (2019 release)



© 2018 Adobe Systems Incorporated and its licensors. All rights reserved.

Developing Structured Applications with FrameMaker FrameMaker Online Manual

If this guide is distributed with software that includes an end-user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end-user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Acrobat, Distiller, Flash, FrameMaker, Illustrator, PageMaker, Photoshop, PostScript, Reader, Garamond, Kozuka Mincho, Kozuka Gothic, MinionPro, and MyriadPro are trademarks of Adobe Systems Incorporated.

Microsoft, Windows, and Windows Vista are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Solaris is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries. UNIX is a trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd. SVG is a trademark of the World Wide Web Consortium; marks of the W3C are registered and held by its host institutions MIT, INRIA, and Keio. All other trademarks are the property of their respective owners.

This product contains either BISAPE and/or TIPEM software by RSA Data Security, Inc.

This product contains color data and/or the Licensed Trademark of The Focoltone Colour System.

PANTONE® Colors displayed in the software application or in the user documentation may not match PANTONE-identified standards. Consult current PANTONE Color Publications for accurate color. PANTONE® and other Pantone, Inc. trademarks are property of Pantone, Inc. © Pantone, Inc. 2003. Pantone, Inc. is the copyright owner of color data and/or software which are licensed to Adobe Systems Incorporated to distribute for use only in combination with Adobe Illustrator. PANTONE Color Data and/or Software shall not be copied onto another disk or into memory unless as part of the execution of Adobe Illustrator software.

Software is produced under Dainippon Ink and Chemicals Inc.'s copyrights of color-data-base derived from Sample Books.

This product contains ImageStream® Graphics and Presentation Filters Copyright ©1991-1996 Inso Corporation and/or Outside In® Viewer Technology ©1992-1996 Inso Corporation. All Rights Reserved.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>).

Certain Spelling portions of this product is based on Proximity Linguistic Technology. ©Copyright 1990 Merriam-Webster Inc. ©Copyright 1990 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 2003 Franklin Electronic Publishers Inc. ©Copyright 2003 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. Legal Supplement ©Copyright 1990/1994 Merriam-Webster Inc./Franklin Electronic Publishers Inc. ©Copyright 1994 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1990/1994 Merriam-Webster Inc./Franklin Electronic Publishers Inc. ©Copyright 1997 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA ©Copyright 1990 Merriam-Webster Inc. ©Copyright 1993 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 2004 Franklin Electronic Publishers Inc. ©Copyright 2004 All rights reserved.

Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1991 Dr. Lluís de Yzaguirre I Maura ©Copyright 1991 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1990 Munksgaard International Publishers Ltd. ©Copyright 1990 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1990 Van Dale Lexicografie bv ©Copyright 1990 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1995 Van Dale Lexicografie bv ©Copyright 1996 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1990 IDE a.s. ©Copyright 1990 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1992 Hachette/Franklin Electronic Publishers Inc. ©Copyright 2004 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1991 Text & Satz Datentechnik ©Copyright 1991 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 2004 Bertelsmann Lexikon Verlag ©Copyright 2004 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 2004 MorphoLogic Inc. ©Copyright 2004 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1990 William Collins Sons & Co. Ltd. ©Copyright 1990 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1993-95 Russicon Company Ltd. ©Copyright 1995 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 2004 IDE a.s. ©Copyright 2004 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. The Hyphenation portion of this product is based on Proximity Linguistic Technology. ©Copyright 2003 Franklin Electronic Publishers Inc. ©Copyright 2003 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1984 William Collins Sons & Co. Ltd. ©Copyright 1988 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1990 Munksgaard International Publishers Ltd. ©Copyright 1990 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1997 Van Dale Lexicografie bv ©Copyright 1997 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1984 Editions Fernand Nathan ©Copyright 1989 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1983 S Fischer Verlag ©Copyright 1997 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1989 Zanichelli ©Copyright 1989 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1989 IDE a.s. ©Copyright 1989 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1990 Espasa-Calpe ©Copyright 1990 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA. ©Copyright 1989 C.A. Stromberg AB. ©Copyright 1989 All rights reserved. Proximity Technology A Division of Franklin Electronic Publishers, Inc. Burlington, New Jersey USA.

Portions of Adobe Acrobat include technology used under license from Autonomy, and are copyrighted.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Notice to U.S. government end users. The software and documentation are "Commercial Items," 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) only as Commercial items and (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. 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.

Contents

.....

.....

PART I: Getting Started

Using Frame Developer Tools	13
The Frame Developer's Kit	13
Choosing the right Frame tools	15
FDK documentation	16
Naming conventions	16
Style conventions	17
Getting Started with FDK	19
Downloading and installing the FDK	20
System requirements	20
Reviewing the sample programs in the samples/hello folder	21
Getting familiar with how the FDK works on Windows	22
Writing FDK clients for Windows	23
Compiling, Registering, and Running FDK Clients	30
Writing an Asynchronous FDK Client	41
Example: adding menus and commands	57
Next Steps	59

PART II: Frame Product Architecture

1 Frame Session Architecture	63
Identifying objects	63
Representing object characteristics with properties	65
FrameMaker product sessions	69
2 Frame Document Architecture	75
Documents	75
Global document information	81
Pages	87
Graphic objects	92
Flows	97
Paragraph Catalog formats	102
Paragraphs	103
Character Catalog formats	108
Condition Formats	112

Text	114
Markers	124
Cross-reference formats	127
Cross-references	129
Variable formats	131
Variables	133
Footnotes	134
Ruling Formats	136
Table Catalog formats	138
Tables	140
Colors	149
Structural element definitions	152
Format rules and format rule clauses	155
Format change lists	157
Structural elements	159
3 Frame Book Architecture	161
What the user sees	161
How the API represents books	162
Creating new books and components	166
Updating a book	167
Using the book error log	172

PART III: Frame Application Program Interface

1 Introduction to the Frame API	177
How the API works	177
Special types of clients	179
Running clients with different FrameMaker product interfaces	181
Creating and running a client	181
A simple example	183
Using old clients with FDK 12	187
2 API Client Initialization	189
Responding to the FrameMaker product's initialization call	189
Initialization types	190
Disabling the API	192
FrameMaker Product Activation by Asynchronous Clients	192
3 Creating Your Client's User Interface 195	
Using API dialog boxes to prompt the user for input	195
Using commands, menu items, and menus in your client	205
Replacing FrameMaker product menus and commands	213
Allowing users to configure your client's interface	213

Using hypertext commands in your client's user interface 215
Responding to user-initiated events or FrameMaker product operations 219
Implementing quick keys 230
Freeing system resources by bailing out 232

4 Executing Commands with API Functions 235
Handling errors 235
Handling messages and warnings 235
Opening documents and books 237
Creating documents 246
Printing documents and books 251
Saving documents and books 253
Closing documents and books 260
Quitting a Frame session 262
Comparing documents and books 262
Updating and generating documents and books 265
Simulating user input 272
Straddling table cells 273
Executing FrameMaker commands 274

5 Getting and Setting Properties 279
What you can do with object properties 279
Getting the IDs of the objects you want to change 280
Manipulating properties 290
Getting and setting session properties 297
Getting and setting document properties 300
Getting and setting graphic object properties 303
Getting and setting paragraph properties 306
Getting and setting book properties 310
Getting and setting FrameMaker properties 311

6 Manipulating Text 319
Getting text 319
Getting and setting the insertion point or text selection 323
Adding and deleting text 333
Getting and setting text formatting 336
Executing Clipboard functions 343

7 Manipulating Asian Text 347
Creating a rubi group 347
Text encodings 348
Using encoding data 350
Inspecting and manipulating encoded text 355
Parsing an encoded string 357
Getting the encoding for a text item 359
Special issues with double byte encodings 359

8	Creating and Deleting API Objects	361
	Creating objects	361
	Deleting objects	381
	Implicit property changes	383
9	Manipulating Commands and Menus with the API	385
	How the API represents commands and menus	385
	Getting the IDs of commands and menus	389
	Determining a session's menu configuration	391
	Arranging menus and menu items	392
	Getting and setting menu item labels	399
	Manipulating expandomatic menu items	401
	Using check marks	402
	Using context-sensitive commands and menu items	402
10	Creating Custom Dialog Boxes for Your Client	407
	Overview	407
	How to create a dialog box	412
	Creating a DRE file	412
	Designing the layout of the dialog box	415
	Setting the properties of the dialog box	419
	Setting the properties of a dialog item	423
	Saving a DRE file	431
	Modeless Dialog Boxes	432
	Testing a dialog box	433
	A simple example	435
	General tips for dialog editing	439
	Summary of keyboard shortcuts	439
11	Handling Custom Dialog Box Events	441
	How the API represents dialog boxes	441
	Overview of using a custom dialog box in your client	444
	Opening dialog resources	448
	Initializing items in a dialog box	449
	Displaying a dialog box	450
	Updating items in a dialog box	451
	Handling user actions in dialog boxes	452
	Closing a dialog box	461
12	Using Imported Files and Insets	463
	Types of imported files and insets	463
	Importing text and graphics	464
	Updating text insets	471
	Client text insets	471
	Writing filter clients	476
	Specifying format IDs and filetype hint strings	480

Associating a file format with signature bytes 492

13 Working with Unicode 505

 Introduction to Unicode Support 505

 Unicode Mode 505

 Compatibility mode 515

 International Components for Unicode (ICU) 523

 Mixed Mode operations 524

 Handling for special characters 524

PART IV: Frame Development Environment (FDE)

14 Introduction to FDE 531

 How the FDE works 531

 How to make your client portable 533

 A simple FDE filter 538

15 Making I/O and Memory Calls Portable 543

 Initializing the FDE 543

 Using platform-independent representations of pathnames 543

 Making I/O portable with channels 547

 Assertion-handler functions 547

 Making memory allocation portable 548

 Error and progress reporting 549

16 FDE Utility Libraries 551

 String library 551

 The string list library 552

 Character library 552

 The I/O library 553

 The hash library 553

 Metric library 555

 MIF data structures and macros 555

 The MIF library 557

 Simple MIF library 558

Glossary 559

PART I



Getting Started

Using Frame Developer Tools

.....

.....

The Frame Developer's Kit

The Frame Developer's Kit™ (FDK) provides tools for developers to enhance the functionality of FrameMaker.

This chapter provides an overview of the FDK and other aspects of FrameMaker that are useful for developers. It also discusses the FDK documentation.

The principal parts of the FDK are:

- Frame Application Program Interface™ (API)
- Frame Development Environment™ (FDE)
- Frame Structure Import/Export Application Program Interface (Structure Import/Export API)

The following sections describe these parts and discuss how you can use them.

Frame API

The Frame API allows you to write C language programs, called *FDK clients*, that can take control of a FrameMaker product session and communicate interactively with the user. With the API, a client can do nearly anything an interactive user can do and more. The API gives a client direct access to the text and graphic objects in documents. The API includes a set of header files, libraries, and makefiles for each supported platform. Here are some examples of the types of clients you can create with the API:

- Grammar checkers
- Bibliography utilities

- Voice control utilities
- Document reporting utilities
- Version control systems for documents
- Table utilities, such as sorting and totaling
- Database publishing packages
- Interfaces to document management systems
- Filters to exchange files between other desktop publishing applications and FrameMaker products

FDE

The Frame Development Environment (FDE) provides platform-independent alternatives to platform-specific I/O, string, and memory allocation schemes. It also provides a variety of utility functions, such as Maker Interchange Format (MIF) writing functions.

Structure Import/Export API

The Structure Import/Export API allows you to write clients that control the import of markup documents into FrameMaker, and control the export of FrameMaker documents to markup.

Other FrameMaker product features for developers

FrameMaker provides other advanced features that are useful for developers. You do not need the FDK to use these features.

MIF

Maker Interchange Format (MIF) is an easily parsed ASCII format that describes a document's text, graphics, formatting, and layout. FrameMaker can save a document or a book to a MIF file, and convert a MIF file back to a document or book, without losing any information.

You can write applications or scripts that convert a MIF file to the format of another desktop publishing package, or convert other formats to MIF.

Here are some examples of things you can use MIF for:

- Sharing files with earlier releases of FrameMaker products
- Converting database files into Frame documents

- Filtering word processor documents into Frame documents

You can find documentation for MIF in the online manuals folder for your FrameMaker installation.

Choosing the right Frame tools

There are often several tools or combinations of tools that you can use to solve a given problem. In particular, you can use the API to perform many of the tasks that MIF and `fmbatch` perform. The tool or combination of tools you should use depends on your needs. Generally, MIF and `fmbatch` are more useful for one-time solutions to small problems, whereas the API is more useful for full-scale applications or applications where interaction with the user is required.

The following table summarizes the advantages and limitations of each Frame tool.

Frame tool or feature	Advantages	Limitations
Frame API	Fast, interactive, and portable; easy to provide a user interface for your applications	Must be compiled
MIF	Can be used by text-processing utilities. It can also be used to provide “backwards” compatibility allowing files to be opened in earlier releases of the product. Third-party MIF creators do not need to write complete MIF. FrameMaker will always write out complete MIF.	Files must be saved as MIF; not interactive

FDK documentation

FDK documentation assumes that you have a thorough knowledge of FrameMaker . For background information on FrameMaker, see your user documentation.

FDK documentation includes the following manuals, which are available in the *doc* folder of your FDK installation.

FDK Programmer's Reference

The *FDK Programmer's Reference* provides FDK reference information, such as error codes and data structure, function, and property descriptions.

FDK Programmer's Guide

The *FDK Programmer's Guide* is the guide you are reading now. It describes how to use the FDK to create clients for FrameMaker. To get the most from this guide, you should be familiar with the C programming language and event-driven programming.

The *FDK Programmer's Guide* is divided into four parts:

- Part I, "Getting Started," provides step-by-step guidance for getting familiar with the FDK.
- Part II, "Frame Product Architecture," provides a conceptual overview of how the API represents sessions, books, and documents.
- Part III, "Frame Application Program Interface (API)," provides instructions for creating API clients.
- Part IV, "Frame Development Environment," provides instructions for making filters and API clients platform-independent.

Naming conventions

To help you identify the structures, constants, and functions defined by the FDK, this manual and the FDK adhere to the following naming conventions:

Type	Naming convention	Example
API error codes	Begin with FE_	FE_NotPgf
API functions	Begin with F_Api	F_ApiGetInt ()

Type	Naming convention	Example
API scriptable function property names	Begin with FS_	FS_NewDoc
FDE functions	Begin with F_	F_StrNew()
Flags used by API functions	Begin with FF_ and all letters are uppercase	FF_UFF_VAR
Initialization constants	Begin with FA_Init	FA_Init_First
Notification constants	Begin with FA_Note	FA_Note_PreFileType
Object property names	Begin with FP_	FP_Fill
Object types	Begin with FO_	FO_Doc
Property value constants	Begin with FV_	FV_Doc_Type_MIF
Typedefs	End with T	MetricT

This manual uses the term *API graphic object* to refer to objects (such as FO_Polygon and FO_TextFrame objects) that the API uses to represent the graphic objects (such as polygons and text frames) that appear on a page.

Style conventions

FDK manuals distinguish between *you*, the developer, and *the user*, the person for whom you write clients.

FDK manuals may use the term *FrameMaker product* to refer to the FrameMaker software, as opposed to the software you write to work with the FrameMaker product.

Structured program interface

FrameMaker 7.0 and later ships with two program interfaces—Structured FrameMaker and FrameMaker. The structured program interface presents menus, icons, and commands for working with structured documents. The FDK includes some functions that only work on structured documents. For example, setting an element range makes no sense in a document that doesn't contain any structure elements. Further, you can specify that an FDK client requires the Structured FrameMaker program interface. For example, assume you specify Structured FrameMaker when you register your client. If a user has your client installed, but is running the FrameMaker program interface (not structured), then his installation of FrameMaker will not initialize your client when it starts up.

The *FDK Programmer's Reference* indicates those FDK functions that apply only to structured FrameMaker documents, as follows:

Structured `F_ApiGetAttributeDefs()`

In this example the word *Structured* appears to the left of the function name, indicating that this function applies only to the content of a structured document. If you register a client to work with the FrameMaker program interface, you should be sure that your client doesn't use any functions identified as *Structured*, otherwise your client may exhibit unpredictable behavior.

Typographic conventions

This manual uses different fonts to represent different types of information.

- What you type is shown in
`text like this.`
- Function names, property names, structure names, returned values, constants, filter names, program names, pathnames, and filenames are also shown in
`text like this.`
- Placeholders (such as those representing names of files and directories) are shown in
text like this.

For example, this represents the name of your working directory:

\Mydir

- Omitted code in source code examples is indicated with ellipses.

For example, the ellipsis in the following code indicates that some of the code necessary to create a complete program is omitted:

```
. . .  
F_ApiAlert((StringT)"Hello world.", FF_ALERT_CONTINUE_NOTE);  
. . .
```

Getting Started with FDK

.....

.....

This Getting Started section is intended to help you get familiar with the basics of FDK. It includes information on creating, compiling, running, and debugging FDK clients. Sample code snippets are provided as pointers that you can build upon and create your own FDK clients.

In this section:

- ["Downloading and installing the FDK"](#)
- ["System requirements"](#)
- ["Reviewing the sample programs in the samples/hello folder"](#)
- ["Getting familiar with how the FDK works on Windows"](#)
- ["Writing FDK clients for Windows"](#)
- ["Compiling, Registering, and Running FDK Clients"](#)
- ["Writing an Asynchronous FDK Client"](#)
- ["Example: adding menus and commands"](#)
- ["Next Steps"](#)

Downloading and installing the FDK

Download the FrameMaker FDK from the FrameMaker Developer Center
<http://www.adobe.com/devnet/framemaker.html>

System requirements

Ensure that your system meets the following requirements:

- Intel Pentium IV
- Microsoft Windows 10, 8.1, or 7
- 1GB of RAM
- 62 MB of available hard-disk space

In addition, you should have Microsoft Visual Studio 2013 installed on the system.

Reviewing the sample programs in the samples/hello folder

The samples folder contains several programs that will help you get started. As an example, here is a code extract from the `samples/hello/hello.c` file:

```
/*
 * Program Name:
 *     hello
 *
 * General Description:
 *     Greet the user at product startup time.
 *
 * Invocation:
 *     Once the client is installed, launch FrameMaker.
 *
 * Install Info (Windows):
 *     Add the following entry (all on one line) to the
 [APIClients]
 *     section of your maker.ini file:
 *
 *     hello=Standard, Greet user at startup,
 *         fdk_install_dir\samples\hello\debug\hello.dll, all
 *
 *     Replace fdk_install_dir with the path of the directory
 *     in which you installed your copy of the FDK files.
 *     Restart maker.
 *
 * Exceptions:
 *     None.
 *
 *****
 ***** /

#include "fapi.h" /* required for all FDK client programs */
#include "fencode.h"

/* Call back invoked at product startup time */
```

```

VoidT F_ApiInitialize(init)
    IntT init;
{
    /* Making it unicode enabled. */
    F_FdeInit();
    F_ApiEnableUnicode(True);
    F_FdeInitFontEncs("UTF-8");
}

```

Getting familiar with how the FDK works on Windows

FDK clients on Windows are not implemented as true Windows clients. They are dynamic link libraries (DLLs) that provide entry points or callback functions, which FrameMaker can invoke.

There are several types of FDK clients:

- A *standard FDK client* is an FDK client that initializes when FrameMaker starts and then waits to respond to specific user actions, such as menu choices.
- A *take-control client* is an FDK client that responds to a special initialization and takes complete control of a FrameMaker session. Many of the effects you can get with this type of client can also be realized by an asynchronous client.
- A *filter* is an FDK client that converts FrameMaker files to or from other file formats. FrameMaker calls a filter when the user attempts to open, import, or save a file with a particular format.
- A *document report* is an FDK client that provides information about a document. The user can start a document report by choosing Utilities>Document Reports from the File menu and selecting the report from the Document Reports dialog box.

When FrameMaker starts, it reads the maker.ini file in the FrameMaker installation directory, and if applicable, the maker.ini file stored in the user's Documents and Settings directory. The [APIClients] section of the maker.ini file contains entries describing the FDK clients to be loaded. FrameMaker then scans the fminit/Plugins directory and subdirectories and loads the FDK clients that have a .dll file extension and valid VERSIONINFO resource information. FrameMaker ignores all other files in the fminit/Plugins directory that do not have a .dll file extension and valid VERSIONINFO resource information.

Writing FDK clients for Windows

How to write an FDK client for Windows

When you write an FDK client, you should do the following for it to compile and run correctly on Windows:

- Include the correct FDK header files in the correct order
- Replace platform-specific functions and data types with FDE equivalents
- Include calls to initialize the FDE if your client calls FDE functions

The following sections discuss these tasks in greater detail.

Including FDK header files

The following table lists the header files you must include in your client in the order in which you must include them.

If you are using	Include
Any FDK function or constant	<code>fapi.h</code>
Any FDE type	<code>fdetypes.h</code>
A specific FDE function	Header file for the function's group (for example, <code>fhash.h</code> for a hash function). For more information, see the function's description in the <i>FDK Programmer's Reference</i> guide.
Any Structure Import/Export API functions	<code>fm_struct.h</code>
Constants for Frame f-codes	<code>fcodes.h</code>

.....
IMPORTANT: You must include the `fapi.h` header file before any other FDK header files. For example, if your client uses API functions and FDE metric utility functions, it must include header files as follows:


```
#include "fapi.h"
#include "fdetypes.h"
#include "fmetrics.h"
```

If you need to include any C library header files or your own header files, include them before the FDK header files.

Adding calls to initialize the FDE

If your client calls FDE functions, it must call `F_FdeInit()` once before it calls the functions. The syntax for `F_FdeInit()` is:

```
ErrorT F_FdeInit(VoidT);
```

To call `F_FdeInit()`, your client must include the `fdetypes.h` header file.

How to write filter clients

You can use filter clients to translate documents from one format to another. FrameMaker invokes an import filter client when it recognizes a file of a particular format or when the file has a registered suffix. It invokes an export filter when you choose a particular format from the Format pop-up menu of the Save As dialog box or save a file using a registered suffix. For example, if you register a suffix for a text import filter and then open a file with that suffix, the Unknown File Type dialog box appears with the appropriate filter preselected.

You must register your filter client before use. For information on registering clients, see ["Compiling, Registering, and Running FDK Clients"](#).

You can also use your filter to import text or graphic files into a document. If you import a file by reference, FrameMaker stores in the document the registered format and vendor ID of the filter used in the import operation. If you import the file by copy, FrameMaker stores the facet name in the document. The information in both these cases ensures that FrameMaker invokes the correct filter for updating the next time you open the document.

.....
IMPORTANT: *If you are writing a filter client, FrameMaker will not fully recognize it unless you include function calls that actually cause the API library to link with your client. To make sure the client links properly, you can include the following as minimal code in your `F_ApiNotification()` function:*


```
. . .  
F_ObjHandleT docId;  
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
```

Identifying your filter

To identify your filter to FrameMaker, you need to supply information in the line that registers the filter. This information identifies the filter on all platforms and identifies the original import filter when reimporting the file. FrameMaker uses several pieces of information that you specify for this purpose:

- The vendor ID is a four-character string describing the provider of the filter.

- The format ID is a four-character string describing the file format of files on which the filter operates.
- The facet name is an arbitrary-length string describing the filter.

For example, assume you create a filter for Windows machines that translates Himyaritic documents to English. You give it the format ID "HIMF" and the vendor ID "FAPI". If you create a document and create a text inset using that filter, FrameMaker stores this information with the inset. The next time you open that document, FrameMaker knows to update the inset with your Himyaritic filter.

FrameMaker reserves the following vendor IDs:

- "FRAM"
- "FFLT"
- "IMAG"
- "XTND"
- "AW4W"
- "ADBE"
- "ADBI"

Your client cannot use these vendor IDs. FrameMaker recognizes FAPI as a valid ID for anyFDK filter client. However, you do not have to use this ID. You can use any other four-characterstring as your vendor ID.

FrameMaker reserves format IDs for the indicated file formats. For a complete list of format Ids, see "[Specifying format IDs and filetype hint strings](#)". FrameMaker does not supply filters for all of these formats. However, to aid in portabilityof your clients, you should not use one of these format IDs unless it is for the specified file format.

Automatic recognition of a file format

Some graphic file formats have signature bytes. Signature bytes are a set of bytes that have a unique value and location in a particular file format. FrameMaker can use signature bytes to identify a graphic file's format.

The documentation for the file format that your graphics filter converts may contain information on the signature bytes for that format. If it does, you can register the signature bytes in the [FormatList] section of the maker . ini file. Each graphic file format description must be on a separate line and must have the following format:

```
n=facet_name start_offset signature_size signature
```

where n is any number, facet_name is the file format's description (also used in the client registration), start_offset is how many bytes from the start of the file the signature begins, signature_size is the size in bytes of the signature, and signature is the hexadecimal value of the signature. You can enclose any of the

arguments in double quotation marks. For example, you can register the file format for MIF with the following:

```
[FormatList]
100="MIF" 0 8 0x3c4d494646696c65
```

where 0x3c4d494646696c65 is the hexadecimal encoding of the characters MIFfile.

Using Windows pathnames

The FDK delimits pathnames with backslashes (\). When you specify a pathname in an FDK function call, follow these rules:

- Follow the drive letter with a colon.
- Don't terminate a pathname that specifies a file with a backslash.

The following table lists examples of files and directories and the pathname strings that specify them.

File or Directory	Absolute Pathname	Relative Pathname
File named myfile.doc on the c: drive	c:\myfile.doc	myfile.doc
Directory named mydir on the c: drive	c:\mydir	mydir

Because the backslash is a special character, you must precede it with another backslash when you specify it in a string. For example, to open a file named c:\myfile.doc with `F_ApiSimpleOpen()`, use the following code:

```
F_ApiSimpleOpen("c:\\myfile.doc", False);
```

Using pathnames returned by FDK functions

Pathnames returned by FDK functions don't end with a backslash, unless they specify rootdirectories, such as c:\.

Using F_PathNameToFilePath()

To specify an absolute pathname when you call `F_PathNameToFilePath()`, you must specify a pathname that includes the drive and begins with the root directory of the drive. If the pathname does not include the drive and begin with the root directory of the drive, `F_PathNameToFilePath()` assumes the pathname is relative.

If you call `F_PathNameToFilePath()` with `anchor` set to `NULL` and you do not specify an absolute pathname, `F_PathNameToFilePath()` adds the currently open

directory or the currently open directory of the specified drive to the pathname. For example, if you specify `c:\myfile.c` for `pathname`, `F_PathNameToFilePath()` generates: `c:\cwd\myfile.c`, where `cwd` is the currently open directory on drive `c:`. If you specify `\\myfile.c` for `pathname`, `F_PathNameToFilePath()` generates: `current_drive:\myfile.c`, where `current_drive` is the current drive.

If you do not set `anchor` to `NULL`, `F_PathNameToFilePath()` constructs the filepath relative to the path specified by `anchor`. If the `pathname` you specify for `pathname` and the filepath you specify for `anchor` are inconsistent, `F_PathNameToFilePath()` ignores `anchor` and constructs the filepath with the currently open directory

Using F_FilePathGetNext()

The function `F_FilePathGetNext()` returns the next file in a specified directory. To do so, this function uses DOS system calls. As a result, since DOS is case-insensitive, the returned `FilePathT` structure uses only uppercase letters. This may not match a `FilePathT` structure you have created.

For example, assume you want to create a filepath and then at some later time process all files in the same directory other than the one you created. You might be tempted to use this code:

```
/* Bad code! */
. . .
/* Create the new filepath */
newpath = F_PathNameToFilePath ("vpg.doc", NULL, F_DosPath);
. . .
DirHandleT handle;
FilePathT *path, *file;
IntT statusp;
pathname = StringT;
handle = F_FilePathOpenDir(newpath, &statusp);
if (handle) {
    pathname = F_FilePathToPathName (newpath);
    while ((file = F_FilePathGetNext (handle, &statusp)) != NULL) {
        /* WRONG! This attempts to compare current file to the one you
        created. */
        if ! (F_StrEqual (pathname, F_FilePathToPathName (file)))
            ProcessFile (file);
        F_FilePathFree (file);
    }
}
/* Bad code! */
. . .
```

The string returned by `F_FilePathToPathName(newpath)` contains the lowercase letters as specified in the earlier call to the function `F_PathNameToFilePath()`. On the other hand, the string returned by each call to `F_FilePathToPathName()` always contains only uppercase letters. Therefore, the call to `F_StrEqual()` never succeeds. Instead of calling `F_StrEqual()`, you should call `F_StrIEqual()`.

Using menus and commands

The following sections describe how to use menus and commands in your FDK client.

Finding FrameMaker menu and command names

The [Files] section of the maker.ini file specifies the location of the menu and command configuration files that list FrameMaker's menus and commands. The following are the default entries in the maker.ini file:

```
MathCharacterFile = fminit\mathchar.cfg
ConfigCommandsFile = fminit\cmds.cfg
MSWinConfigCommandsFile = fminit\wincmds.cfg
ConfigMathFile = fminit\mathcmds.cfg
ConfigMenuFile = fminit\maker\menus.cfg
ConfigCustomUIFile = fminit\customui.cfg
```

The following table lists the menus and commands each file contains.

Menu or Command File	Contents
MathCharacterFile	Special math characters
ConfigCommandsFile	Basic commands
MSWinConfigCommandsFile	Windows-specific commands
ConfigMathFile	Math commands
ConfigMenuFile	Standard menus
ConfigCustomUIFile	Custom menus

Using FDK functions that write to FrameMaker console

The following functions write output to the FrameMaker console on Windows:

- F_ApiPrintFAErrno()
- F_ApiPrintOpenStatus()
- F_ApiPrintPropVals()
- F_ApiPrintSaveStatus()
- F_Printf() with Channel set to NULL
- F_Warning()

For descriptions of these functions, see the *FDK Programmer's Reference* guide. As with printf(), the F_Printf() function does not automatically print a line feed ("\n") after the output. If you don't end the output with "\n", the next call to one of the functions listed above begins printing on the last line printed by the F_Printf() call.

Using platform-dependent session properties

Session (FO_Session) objects have the following platform-dependent properties:

Property	Value
FP_FM_BinDir	Pathname of the bin directory in the FrameMaker installation directory
FP_FM_CurrentDir	Pathname of the FrameMaker installation directory
FP_FM_HomeDir	Pathname of the FrameMaker installation directory
FP_FM_InitDir	Pathname of the fminit directory in the FrameMaker installation directory
FP_HostName	Host name specified for PCName in the maker.ini file
FP_OpenDir	Pathname of the FrameMaker installation directory
FP_Path	Path specified by the \$PATH environment variable
FP_TmpDir	Directory specified by the \$TEMP environment variable
FP_UserHomeDir	Pathname of the FrameMaker installation directory
FP_UserLogin	The user name under which FrameMaker is registered
FP_UserName	The user name under which FrameMaker is registered

Although the values of some of these properties specify directory pathnames, they are not terminated with a backslash.

Compiling, Registering, and Running FDK Clients

This section describes how to compile, register, and run FDK clients on Windows. It also briefly explains how to debug your FDK clients.

Compiling FDK Clients

The following sections describe how to compile FDK sample clients and your own clients.

Supported compilers

To compile FDK clients for Windows, you must use Microsoft Visual Studio 2010.

Compiling and registering sample clients in Microsoft Visual Studio 2010

To compile and register a sample FDK client in Microsoft Visual Studio 2010, follow these steps:

1. Start Microsoft Visual Studio 2010
2. Open the Project and then choose the solution file for one of the sample clients. For example, to compile the `aframes` sample client, choose `fdk_install_dir\samples\aframes\aframes.sln`, where `fdk_install_dir` is the pathname of the directory in which the FDK is installed.

NOTE: The project settings for the sample clients have relative paths to the FDK lib and include files already specified. If you open a sample project from its location in the FDK installation, these paths will be valid. If you move the sample client to a different location, you may need to specify new paths for the include and lib files. For more information, see ["Compiling and registering your own FDK clients"](#).

3. Use the build utility to build the client. Choose Rebuild Solution from the Build menu. Microsoft Visual Studio 2010 compiles your code into a DLL file named `project.dll` in the debug subdirectory of your client directory, where `project` is the name of the sample project. For example, the `aframes` sample client compiles into `debug\aframes.dll`.
4. Register the sample client.

Each of the following sample clients includes a VERSIONINFO resource, and you register each by placing the DLL file in the Plugins folder:

- `pickfmts`
- `elemutils`
- `dialog`

Because the remaining sample clients do not include a VERSIONINFO resource, you must register them in the `maker.ini` file. For more information see ["Registering clients in the FrameMaker maker.ini file"](#).

Running the sample FDK clients

It is best to store client DLL files in the FrameMaker Plugins folder (`install_dir\FramerMaker<version>\fminit\Plugins`), or in a folder below it. If you register your clients via the `VersionInfo` resource, you must store them in this way. When you register a client in the `.ini` file, you can specify any location for the DLL file.

After you have compiled and registered a sample FDK client, start FrameMaker to test the client. Some of the sample clients add menus and commands to the FrameMaker menus. For example, if you have compiled and registered the sample client described in

["Introduction to the Frame API"](#), a menu named API appears on the FrameMaker menu after you start FrameMaker. To test the commands on this menu, open or create a document, and choose each of the commands.

Compiling and registering your own FDK clients

To compile and register one of your own FDK clients, follow the instructions in this section.

Compiling and registering the client

To compile and register the FDK client, follow these general steps:

1. Create a project directory for your FDK client project.
2. Start Microsoft Visual Studio 2010 and create a new Win32 Dynamic- Link Library project.

Choose New and then project from the File menu. The New dialog box appears. Select Visual C++ Projects and then Win32 from Project Types. Select Win32 Project, type your client's name in the Name field and then click OK. Win32 application wizard appears.

Click on Application Settings, select DLL from Application Type and Empty project from Additional Options.

3. Create or place your source files in the project directory, then add those files to your project.
4. (Optional) Create a resource for any custom dialog boxes. If your client contains custom dialog boxes, you need to create a resource for them. For instructions, see ["Handling Custom Dialog Box Events"](#).
5. (Optional) Create a VERSIONINFO resource. Including a VERSIONINFO resource is one method for registering a client. For more information on registering clients, see ["Registering FDK Clients"](#).
6. Choose Properties from the Project menu to display the Properties Pages dialog box.

In the Properties Pages dialog box, choose General . Set Use of MFC field to Use Standard Windows Libraries.

.....
IMPORTANT: *If you don't set the Use of MFC field to "Use Standard Windows Libraries", your client will not link correctly.*

7. Set your project's C/C++ Language options.

In the Property Pages dialog box, choose C/C++.

Choose Code Generation and choose 8 Byte or default from the Struct Member Alignment pull down menu. 8 bytes is also the default value for this field.

.....
IMPORTANT: *If you don't set the Struct Member Alignment to 8 Bytes, your client may cause unexpected runtime errors.*
.....

With Code Generation still selected, choose Single-Threaded from the Runtime Library popup list. The FDK ships in a single-threaded version. By default, the project sets this option to Multi-threaded. Compiling the FDK with a multi-threaded runtime library produces the following warning:

```
defaultlib "LIBC" conflicts with use of other libs;
```

.....
IMPORTANT: *For Version 7.0 and later of the FDK, it is important that you make this setting. Earlier versions of the FDK did not use symbols that conflicted with the multi-threaded runtime library. However, for version 7.0 and later the FDK and the Structure Import/Export API use conflicting symbols.*
.....

In the General page under C/C++ language options, add path to the FDK include files in Additional Include Directories field. You can specify an absolute path or a relative path. For example, the Property Pages for the sample clients all use the following relative path:

```
..\..\include
```

8. Set your project's Linker options:

In the Property Pages dialog box, choose the Linker page.

Choose General and then Input.

Add the FDK libraries `fdk.lib`, `api.lib`, and `fmdbms32.lib` to the additional dependencies field.

If you are compiling a structure import/export client, be sure to also link the Structure Import/Export API library. For more information, see ["Linking the Structure Import/Export API library"](#).

.....
IMPORTANT: *If your client includes custom dialog boxes, you must add /section:.rsrc,w to the Project Options. For more information, see ["Compiling clients with custom dialog boxes"](#).*
.....

In the Category field, choose Input, then add the path to the FDK lib files in the Additional library path field.

You can specify an absolute path or a relative path. For example, the project settings for the sample clients all use the following relative path:

```
..\..\lib
```

As an alternative, you can specify access to the FDK include and lib directories for the Development Environment 2003. To do this, choose Tools > Options to display the Options dialog box. Select Project and then VC++ Directories, and enter the paths to the FDK include and lib directories for Include files and Library files.

9. Use Microsoft Visual Studio 2010 build utility to build your client.

Choose Rebuild All from the Build menu. Visual Studio compiles your code into a dynamic link library file with the name you typed in the New dialog box. It puts this library file into the debug subdirectory of your client directory.

10. Register the client.

You can register the client by using either of these two methods:

- As mentioned in step 5, create and include in your client's project a VERSIONINFO resource that contains information about the client, and copy or move the compiled client into the `fminit/Plugins` directory.
- Add an entry for your client in the `[APIClients]` section of the `maker.ini` file in the FrameMaker directory.

For more information on registering clients, see ["Registering FDK Clients"](#).

Using custom dialog boxes

The FDK samples include a template document for designing custom dialog boxes. You open this document in FrameMaker and edit it with the FrameMaker graphic tools and commands.

When you save a custom dialog box in a Windows version of FrameMaker, it generates two Windows resource definition files, a `.dlg` file and a `.xdi` file.

- The `.dlg` file is a text file containing resource statements. These statements are standard Windows descriptions of the dialog box and its controls.
- The `.xdi` file is a text file containing a user-defined resource statement. This statement contains data used by FrameMaker to manipulate the dialog boxes.

When creating the `.dlg` and `.xdi` files, FrameMaker uses the name of the `.dre` file (without the extension) to name the files and the actual dialog resource. For example, when saving the file named `mydlg.dre`, FrameMaker creates the resource description files `mydlg.dlg` and `mydlg.xdi`. Both files describe the dialog resource named `mydlg`.

To compile the `.dlg` and `.xdi` files in your `dll` you must create a resource for the project, and provide directives to include these files in the resource. In the process of

compiling the client, these resource definition files are compiled into a single resource file (.rc). This resource file is linked to your client. To set up the resource definition files to be compiled, follow these general steps:

1. Start Microsoft Visual Studio 2010.
2. If one doesn't already exist for the project, create a resource script. Choose Add New Item from the File menu. The Add New Item dialog box appears. Choose Resource File.
3. Include the resource description files generated by FrameMaker.

Choose Resource Includes from the Edit menu. The Resource Includes dialog box appears.

In the Compile-Time Directives field, type #include statements to include the resource description files. For example, suppose you create two custom dialog boxes named pgftag.dre and chartag.dre. When FrameMaker saves these files it also creates the files pgftag.dlg, pgftag.xdi, chartag.dlg, and chartag.xdi.

To include these files in the resource script, type the following in the Compile-Time Directives field:

```
#include "pgftag.dlg"  
#include "pgftag.xdi"  
#include "chartag.dlg"  
#include "chartag.xdi"
```

Compiling clients with custom dialog boxes

If your FDK client uses custom dialog boxes, you need to specify a special link option before compiling it:

1. In Microsoft Visual Studio, choose Project->Properties. This displays the Project Properties dialog box.
2. Choose Linker and then Command Line.
3. Add the following option to the Additional Options field:

```
/section:.rsrc,w
```

This link option makes the dialog resources writable. If you do not specify it before compiling, your FDK client may exit unexpectedly when it attempts to display a custom dialog box.

4. Repeat steps 3 for each target in your project.

Making adjustments to custom dialog boxes

Since the .dlg files produced by FrameMaker are text files containing resource statements, you can open these files as resources. You can use the built-in tools for

dialog editing to view, adjust, and test the dialog box. Because you are modifying the `.dlg` file but not the `.xdi` file, you should not make major changes to the dialog box (for example, do not add new items to the dialog box). If you do, the description in the `.dlg` file will not match the description in the `.xdi` file.

Linking the Structure Import/Export API library

To link the Structure Import/Export API library on Windows follow these steps:

1. In Microsoft Visual Studio 2010, open your client's project.
2. Choose Properties from the Project menu to display the Properties Pages dialog box.
3. In the Property Pages dialog box, click on Linker and then Input.
4. Add the Structure Import/Export API library `struct.lib` and the resource `fmstruct.res` to the Additional Dependencies field. Add `struct.lib` before `fdk.lib`, and add `fmstruct.res` to the end of the Object/Library Modules field.
5. Add the following link option to the 'Additional Options' field in 'Command Line' property page:
`/section:.rsrc,w`

.....
IMPORTANT: *This link option is required for some of the dialog boxes that are internal to the structure import/export functionality in FrameMaker. Without this link option, your client may crash when it interacts with these dialog boxes.*

Registering FDK Clients

For FrameMaker to recognize your client, you must register it on the system on which you intend to run it. When registering your client, you can name it anything you like, although the name cannot contain spaces. Also, you should not use a name that is already used by one of the clients that ships with FrameMaker.

To register your client, you add an entry for your client in the `[APIClients]` section of the `maker.ini` file in the FrameMaker directory. The `[APIClients]` section of the `maker.ini` file lists the FDK clients to load when FrameMaker starts. For more information on registering your client using the `maker.ini` file, see ["Registering clients in the FrameMaker maker.ini file"](#).

Registering clients in the FrameMaker maker.ini file

You can register a client is by adding an entry for the client in the FrameMaker `maker.ini` file. The `[APIClients]` section of the `maker.ini` file lists the FDK clients

to load when FrameMaker starts. Each client description must be on a separate line and cannot contain line breaks. Clients that are not filters use the following format:

```
client = type, description, DLL_file, mode
```

where

For this statement	Specify
client	the client's name
type	the type of client—valid types for clients other than filters are Standard, TakeControl, and DocReport
DLL_file	the pathname of the client's DLL file—can specify a full pathname or a relative pathname based on the FrameMaker installation directory.
mode	whether the client can run with FrameMaker in unstructured or structured mode. This field can be one of maker, structured, or all. The mode field is required.

The fields in this line are separated by a comma and zero or more spaces. For example, if you have compiled the aframes sample client into

c:\fdk\samples\aframes\debug\aframes.dll, and you want to register it with FrameMaker, add the following to the maker.ini file in the FrameMaker installation directory (without any line breaks):

```
AFrames=DocReport,Anchored Frames Report,c:\fdk\samples\aframes\debug\aframes.dll, all
```

If the client is a filter, register it with the following line:

```
client = type, facet_name, format_id, vendor_id, display_name, description, DLL_file, mode, suffix
```

where the variables are:

For this statement	Specify
type	One of: <ul style="list-style-type: none"> • TextImport • GFXImport • Export • FileToFileTextImport • FileToFileTextExport • FileToFileGFXImport • FileToFileGFXExport
facet_name	the name of the file format supported by the client.
format_id	a four-character string that identifies the file format
vendor_id	a four-character string that identifies the client's provider
display_name	the filter name to display in in dialog boxes when opening or saving a file of the given format. This name must match the client name.
description	a description of the client that appears when you choose About
DLL_file	the pathname of the client's DLL file
mode	whether the client can run with FrameMaker in unstructured or structured mode. This fields can be one of maker, structured, or all. The mode field is required.
suffix	the filename extension of the file type that the client filters

For information on format and vendor IDs, see ["How to write filter clients"](#). For example, assume you have a graphics import filter for the CGM format that uses ACGM as its facet name, has its executable stored in acgmflt.dll, and should be invoked on files with the suffix cgm. You can register this filter with this line:

```
ACGMFILTER=GFXImport,ACGM,CGM,FAPI,ACGMFILTER,acgmflt.dll,all,cgm
```

Specifying no description for a client

When you register your client by using the FrameMaker maker.ini file, and you don't want to specify a description, enter a space in the description field. For example:

```
client= Standard, ,c:\clients\myclient\debug\myclient.dll, all
```

The description field must contain at least one character. If no characters appear between the commas delimiting the description field, your client will not be registered.

Running FDK Clients

When FrameMaker starts, it reads the `maker.ini` file. The `[APIClients]` section of the `maker.ini` file contains entries describing the FDK clients to be loaded. FrameMaker then scans the `fminit/Plugins` directory and subdirectories and loads the FDK clients that have a `.dll` file extension and valid `VERSIONINFO` resource information.

FrameMaker ignores any files in the `fminit/Plugins` directory and subdirectories that do not have a name with the `.dll` extension, or do not contain valid `VERSIONINFO` resource information.

For information on how FrameMaker starts a client, see "API Client Initialization" in the *FDK Programmer's Guide*.

Compatibility between FDK and FrameMaker product releases

To ensure your existing Windows clients are compatible with release 11 of FrameMaker, you should recompile them. It is possible to run a client compiled in an earlier version of the FDK with FrameMaker 11, as long as the client does not use any functions or properties that have changed. However, it is recommended that you recompile your clients with the newer version of the FDK as soon as possible.

Disabling FDK clients

To disable all FDK clients, edit the following line in the `maker.ini` file in the FrameMaker installation directory, or in the version of the `.ini` file that is stored in the user's Documents and Settings directory:

```
API=On
```

Replace `On` with `Off`. The next time you start FrameMaker, no FDK clients will be started.

.....
IMPORTANT: *Some FrameMaker features, such as the Word Count document report, Save As HTML, or import and export of XML are implemented as FDK clients. If you disable all FDK clients, these features will not be available.*
.....

Debugging FDK Clients

You debug your client as part of the FrameMaker executable. The FrameMaker executable is not compiled with debugging information, so you don't have access to any symbols within FrameMaker.

To use Microsoft Visual Studio to debug your client as part of the FrameMaker executable, follow these general steps:

1. Start Microsoft Visual Studio 2010.
2. Open your client's project and add breakpoints.
3. Select Project->Properties and then Debugging page. Go to Command Field and add the path to FrameMaker executable.pen the FrameMaker executable.
For example, if FrameMaker is installed in

```
c:\Program Files\Adobe\FrameMaker10,
```

then to open its executable, open

```
c:\Program Files\Adobe\FrameMaker10\FrameMaker.exe.
```

4. From the Build menu select Configuration Manager. Highlight the Debug Project Configuration.
5. From the Debug menu, choose Start. Alternately, if you have already started the debugger for your program, from the Debug menu choose Restart. If FrameMaker isn't able to load your client, it displays the following error message in an alert box:

```
File Error: Cannot find client_name.dll
```

FrameMaker may not be able to load your client for the following reasons:

- The client is not located in the fminit/Plugins directory or subdirectories, or does not have a name with the .dll extension.
- The client's VERSIONINFO resource information is missing or invalid.
- The maker.ini file doesn't specify the correct full pathname for your client's DLL.
- The FrameMaker release is incompatible with the FDK release that you used to compile the client.

To check that your FDK client has control, you can have it display a string in the status bar of the document or book window. For more information, see the descriptions of FO_Book and FO_Doc in the *FDK Programmer's Reference* guide.

Writing an Asynchronous FDK Client

This section describes how to create asynchronous clients on Windows, and provides instructions for compiling and running a sample asynchronous client. Before writing an asynchronous API client you should be familiar with both the FrameMaker FDK and Windows API programming.

The purpose of many FDK clients is to modify FrameMaker in some way, such as by changing or adding functionality. In these applications the main goal of the resultant application is still for the end user to use FrameMaker.

A different kind of application is one that uses FrameMaker to support some aspect of the application's functionality, but in which use of FrameMaker is not the goal. For example, you might create a data base and want to use FrameMaker to print catalogs from it. In this case, your application runs primarily independently of FrameMaker, but calls FrameMaker (possibly as a child process) during some part of its operation.

The FDK allow you to create asynchronous applications that control a FrameMaker process. Even though the main purpose of the application may not be to run FrameMaker, this chapter refers to such an application as an FDK client, since it calls FDK functions.

An asynchronous client does not run as part of the FrameMaker process nor as a child process. Instead, it is its own application in a separate process, communicating with a FrameMaker process via Microsoft RPC (Remote Procedure Calls). You should be aware of some consequences of this difference:

- An asynchronous client can be started independently of any FrameMaker product. It can be an EXE, or a DLL of some EXE other than FrameMaker.
- It must have its own main() function.
- You can use MFC or any other application framework to develop an asynchronous client.
- An asynchronous client can run on a machine other than that running the associated FrameMaker process.

End user installations

To run asynchronous clients, the executable applications or the DLL files must be installed correctly. An EXE can be installed wherever the user wants. A DLL that is a plugin for another application must be installed correctly for that application. A DLL that is a plugin for FrameMaker must be installed in the appropriate Plugins directory, or its path must be specified in the maker.ini file.

The user also must have the following files installed in his or her FrameMaker installation directory, at the same level as the FrameMaker application:

- `afmfdk.dll`
- `fmrncInt.exe`

In addition, the user must have the following entries in the `maker.ini` file:

```
[Files]
MarshallingDLL = afmfdk.dll
RunWrappedPlugin = fmrncInt.exe
. . .
[Preferences]
ExecutablePlugins = EXE
WrappedPlugins = DLX
PluginExtensions = DLL, DLX, EXE
```

The `[Preferences]` entries tell `FrameMaker` which filename extensions are valid for different types of clients.

- `PluginExtensions` must list extensions for all the files you want to be loaded as clients of any type.
- `ExecutablePlugins` lists extensions for clients that are built as executables which run outside of the `FrameMaker` process.
- `WrappedPlugins` lists extensions for clients that are built as DLLs, but will run in an address space that is external to the `FrameMaker` process. Such a client uses `fmrncInt.exe` to wrap its DLL and runs in the `fmrncInt.exe` address space.

Note that you can substitute other extensions for the ones shown in the example above. For more information, see ["Types of asynchronous clients"](#).

Registering asynchronous clients

You can register asynchronous clients just as you register other clients; you can store the registration data in the client's `VersionInfo` resource, or you can make an entry in the `maker.ini` file for `FrameMaker`. Additionally, your client can pass an `F_PropValsT` structure to `F_ApiWinConnectSession()` that is a list of registration data.

`F_ApiWinConnectSession()` is defined as:

```
F_ApiWinConnectSession(const F_PropValsT *connectProps,
ConStringT hostname, const struct _GUID *service);
```

You can include the following properties in `connectProps`:

This property	corresponds to this statement in a client's VERSIONINFO resource
FI_PLUGIN_NAME	the name of the client.
FI_PLUGIN_TYPE	the type of client
FI_PLUGIN_PRODUCTS	specifies structured or unstructured FrameMaker, using the names of FrameMaker products this client supports—use a space-delimited string with one or both of <code>Maker</code> and <code>MakerSGML</code>
FI_PLUGIN_FACET	the name of the file format supported by the client (filters, only)
FI_PLUGIN_FORMATID	a four-character string that identifies a file format (filters, only).
FI_PLUGIN_VENDOR	a four-character string that identifies the client's provider.
FI_PLUGIN_SUFFIX	the filename extension of the file type that the client filters (filters, only).
FI_PLUGIN_INFORMAT	the file format for the file to filter (filters, only)
FI_PLUGIN_OUTFORMAT	the file format for the resulting file (filters, only)
FI_PLUGIN_DESCRIPTION	a description of the client that appears when you choose About
FI_PLUGIN_PRODUCTNAME	the name by which customers know your client.

If `connectProps` is NULL, the FrameMaker process uses the client's `VersionInfo` resource or the entries in the `maker.ini` file. If the client has no registration information in any of these sources, the FrameMaker process registers it as a standard client.

Types of asynchronous clients

Asynchronous clients can be executable applications (EXE), dynamically linked libraries (DLLs) that are a part of another application, or DLLs that are plugins for FrameMaker (wrapped plugins).

Asynchronous EXE applications

An EXE can be either a console application or a Windows application. After connecting with the FrameMaker process, the EXE application passes calls to FrameMaker through `afmfdk.dll`.

.....
IMPORTANT: *Because they don't have a Windows message processing loop, console applications cannot handle notifications from the FDK. For example, this means a console application cannot process commands from menus it adds to FrameMaker. Nor can it process notifications such as `FA_Note_PreOpenDoc` or `FA_Note_PreSaveDoc`.*

Asynchronous DLLs

A DLL that is part of another application can call `F_ApiStartUp()` to make a connection with a FrameMaker process. For example, you could write a plugin for Acrobat Exchange that writes the data from Acrobat Forms to a FrameMaker document. In that case, the DLL communicates with the FrameMaker process, as a part of its parent EXE, via `afmfdk.dll`.

A DLL that runs as a wrapped plugin for FrameMaker runs in its own memory space. After connecting with the FrameMaker process, the DLL invokes `fmrnc1nt.exe` to run as a wrapper for the DLL. The wrapped DLL then communicates with FrameMaker via `afmfdk.dll`, as though it is an EXE.

Registering multiple FrameMaker processes as servers

When you first run FrameMaker, it registers itself in the system registry as the default instance of the FrameMaker instance on that machine. By default, asynchronous clients connect to this instance.

You can register multiple instances of the FrameMaker process, each with a unique entry in the system registry. Then you can use these processes as a bank of servers, and your client can choose among them when making a connection.

You identify a FrameMaker process as a server by its entry in the system registry. The entry can specify:

- A name to identify the GUID for that specific process.
- Whether the process starts up when called by a client, or whether it must already be running before the client can connect to it.

To register a process, you start FrameMaker with specific commandline options. This creates an entry in the system registry for the machine on which you start FrameMaker.

To start FrameMaker with commandline options:

1. Choose Run from the Start menu.

The Run Application dialog box appears.

2. In the text box, type the full pathname of the FrameMaker.exe file, followed by the commandline options. Alternately, you can start FrameMaker from a DOS Command Prompt window. For example, type `FrameMaker_path\FrameMaker10 /option`, where `FrameMaker_path` is the install path for the version of FrameMaker you want to run, and `/option` is one or more of:

- `progid:process_name`

where `process_name` is a name you provide. This option registers a name for the FrameMaker process.

- `auto`

This option allows the FrameMaker process to automatically start up if it isn't running when another process calls it.

- `noauto`

This option disallows automatic start-up.

This creates an entry in the system registry for the machine on which you started FrameMaker.

Registering a name for a FrameMaker process

To specify a name for the process, use the `/progid` option. For example, type `FrameMaker_path\FrameMaker10 /progid:MyProcess.Api1`, where `FrameMaker_path` is the install path for the version of FrameMaker you want to run. This establishes a name, `MyProcess.Api1`, for the process.

When you start FrameMaker with no `/progid` option, you create system registry entry with the default name of `FrameMaker.API.1`.

Asynchronous clients running locally on the host can refer to processes by their names. In this way, your client can choose which process to run for a given task.

.....
IMPORTANT: *Clients connecting to a remote host cannot use the process name to connect to a FrameMaker process. Instead, they must use the GUID for that process, as it is specified in the system registry.*
.....

Registering automatic start-up for a process

If the FrameMaker process is not running, an asynchronous client can still call it. If the process is so registered, it will start up when the client calls it. Alternatively, you can register the process in a way that does not allow automatic start-up.

To register the process for automatic start-up, use the `/auto` option. To disallow automatic start-up, use the `/noauto` option. For example, type `FrameMaker_path\FrameMaker7.2 /progid:MyProcess.Api1 /auto`, where `FrameMaker_path` is the install path for the version of FrameMaker you want to run. This establishes a process named `MyProcess.Api1`, which will start automatically when an asynchronous client calls it.

Running asynchronous clients on remote hosts

With systems that support DCOM, you can run a client on one machine (the client machine), connected to a FrameMaker process on another machine (the host machine). To accomplish this, you make use of the DCOM services provided with your operating system. Also, both machines must be in the same domain, and the same user must have the accounts on both machines.

For an asynchronous client to connect to a FrameMaker process on a remote host:

1. Register the FrameMaker process as a server process on the host machine.

This establishes entries on the host machine's system registry for the FrameMaker processes you want to run as servers. For more information see "[Registering multiple FrameMaker processes as servers](#)".

2. Run `dcomcnfg` on the host machine to configure DCOM accessibility for each process you want to run as a server.

This enables DCOM connections to the FrameMaker server processes that are registered on the host machine.

3. Run `dcomcnfg` on the client machine to configure its DCOM accessibility.

This enables the client machine to connect to the host machine via DCOM.

Enabling DCOM for the server processes on the host

To enable DCOM for a FrameMaker process on the host machine:

1. Choose Run from the Start menu.

The Run dialog box appears.

2. In the Run dialog box, type `dcomcnfg`

The DCOM Configuration Properties service application appears.

3. Select the Default Properties tab and click Enable Distributed COM on this Computer.
4. In the Applications list box, double-click the FrameMaker process you want to enable, then set the appropriate security options.

5. Click the Security tab and make sure Use default configuration permissions is turned on.
6. Apply any other settings to the FrameMaker process or your computer that are appropriate for your network configuration. You should check with the system administrator to ensure the options you set are compatible with his administration procedures.
7. Click OK.

Enabling DCOM for client machine

To enable DCOM on the client machine:

1. Choose Run from the Start menu.

The Run dialog box appears.

2. In the Run dialog box, type `dcomcnfg`
3. The DCOM services application appears.
4. Select the Default Properties tab and click Enable Distributed COM on this Computer.
5. Apply any other settings to your computer that are appropriate for your network configuration. You should check with the system administrator to ensure the options you set are compatible with the administration procedures.
6. Click OK.

To find more information on DCOM see the Windows Online Help.

Connecting with a FrameMaker process

Asynchronous clients connect with a FrameMaker process by calling `F_ApiStartUp()` or `F_ApiWinConnectSession()`. When connecting to a process on a local host, FrameMaker does not have to be registered as a server. For a process on remote host, your client must know the GUID for that process.

A machine may have more than one FrameMaker process running at a time. In that case, the processes must be registered as servers, and they should be registered with a name for each process. For information about registering FrameMaker processes as servers, see ["Registering multiple FrameMaker processes as servers"](#).

Asynchronous clients use COM to communicate with FrameMaker processes. If any FDK call returns `FE_Busy`, then you probably need to register a message filter. When using COM, an application should always register a message filter. If your code calls `F_ApiStartUp()` or `F_ApiWinConnectSession()` before initializing COM, these routines automatically initialize COM and register a message filter. However, if you initialized COM before calling these routines, they assume your application already

registered a message filter. If your application initializes COM but does not register a message filter, be sure to call `F_ApiWinInstallDefaultMessageFilter()`.

Connecting to the default process on a local host

You use `F_ApiStartUp()` when the desired FrameMaker process is running on the local machine. For example, a DLL that is a FrameMaker plugin calls `F_ApiStartUp()`. In that case, the FrameMaker process that invokes the DLL identifies itself by passing a globally unique identifier (GUID) via the `FMGUID` environment variable. Likewise, if you want an EXE to connect locally to the currently active FrameMaker process, use `F_ApiStartUp()`.

The following call makes this connection:

```
F_ApiStartUp(NULL);
```

For more information, see `F_ApiStartUp()` in the *FDK Programmer's Reference guide*.

Connecting to a named process on a local host

To connect to a named process on a local machine, you need to convert the process name to a GUID. Then you can pass that GUID to `F_ApiWinConnectSession()` to initiate communication between your client and the FrameMaker process.

Note that `F_ApiStartUp()` makes a reliable connection only when the desired FrameMaker process is the only FrameMaker process running on the local host. If no FrameMaker process is running, `F_ApiStartUp()` will not work. Also, if more than one process is running,

`F_ApiStartUp()` cannot determine which process will finally connect with your client. To choose one of many FrameMaker processes on a local host, you should have all of the processes registered as servers on that host.

If you have registered the process as a named server, and your client is connecting to it on a local host, you can use the Win32 API to get the GUID associated with that name. Then you pass the GUID to `F_ApiWinConnectSession()`.

The following example uses the Win32 API function `CLSIDFromProgID()` to get the GUID for a process named `MyProcess.Api1`. It then calls `F_ApiWinConnectSession()` to connect to the process. Note that you need a

Unicode string for the process name. The example uses the Win32 API call, `MultiByteToWideChar()` to convert a string to Unicode.

```
#define WBUFLEN 512
OLECHAR progStr;
CLSID serviceId;
StringT myProcess = F_StrCopyString("MyProcess.API.1");
. . .
progStr = (OLECHAR*)malloc( WBUFLEN*sizeof(wchar_t) );
MultiByteToWideChar(CP_ACP, 0, (char *)opt_progid, -1, progStr,
WBUFLEN );
if(CLSIDFromProgID(progStr, &serviceId))
    F_ApiConnectWinSession(0, 0, &serviceId);
. . .
```

Note that `F_ApiWinConnectSession()` takes three parameters. In the first parameter you can pass a list of properties that correspond to the entries you provide when registering a FrameMaker client.

The second parameter is for the address of a remote host, when making a connection to a remote host. If this parameter is `NULL` or `0`, `F_ApiWinConnectSession()` connects to the local host.

The third parameter specifies the desired FrameMaker process on the host machine. If this parameter is `NULL` or `0`, `F_ApiWinConnectSession()` uses the value of the `FMGUID` environment variable on the specified host.

For more information, see `F_ApiWinConnectSession()` in the *FDK Programmer's Reference* guide.

Connecting to a remote host

To connect to a remote machine, you need the address of that machine. Once you have the address, you can call `F_ApiWinConnectSession()` to initiate communication between your client process and the FrameMaker process on the host machine. The following call makes this connection to the currently running FrameMaker process on the remote host:

```
F_ApiWinConnectSession(0, remote, 0);
```

where `remote` is the address of the remote host.

The above call only works when the desired FrameMaker process is the only FrameMaker process running on the remote host. If no FrameMaker process is running, this will not work. Also, if more than one process is running, you cannot predict which process will finally connect with your client.

To choose one of many FrameMaker processes on a remote host, you should have all of the processes registered as servers on that host.

To choose a registered process, you must know the GUID for that process ahead of time; you pass that GUID to `F_ApiWinConnectSession()`. Assuming you have specified a GUID in `serviceId`, the following call connects to a specific process on the remote host:

```
stringT remote;`
CLSID serviceId;
. . .
F_ApiWinConnectSession(0, remote, &serviceId);
```

where `remote` is the address string of the machine that is running the FrameMaker process.

How to write an asynchronous FDK client

To write an asynchronous client that communicates with FrameMaker, you proceed as you would for any C application, providing a `main()` function and adding whatever functionality you need.

A Windows client can get control of a FrameMaker process by invoking `F_ApiCallClient()` to call itself. For the duration of the notification, that is while the client is processing the resulting callback, the client has exclusive control of FrameMaker. At some point in its processing, your client needs to communicate with a FrameMaker process. To do so, it follows these general steps:

1. Connect to the FrameMaker process.

To connect to a local host, use `F_ApiStartUp()` or `F_ApiWinConnectSession()`. To connect to a remote host, use `F_ApiWinConnectSession()`. For information about connecting to FrameMaker processes, see ["Connecting with a FrameMaker process"](#). For information about the functions to connect to FrameMaker processes, see `F_ApiWinConnectSession()` and `F_ApiStartUp()` in the *FDK Programmer's Reference guide*.

2. Depending on your client, wait for requests from FrameMaker or perform some operations using FrameMaker.

Once connected to a running FrameMaker process, your client can use the FDK to control the FrameMaker process, or receive notifications from it. However, bear in mind that console programs cannot handle notifications from the FDK. (This is because console programs do not have a Windows message processing loop; applications running in console programs must not request notifications.)

Note that a client can take exclusive control of the FrameMaker process by requesting notification for `FA_Note_ClientCall` and then calling itself via `F_ApiCallClient()`.

While handling the notification, no other clients can take control of the FrameMaker process.

3. When done, disconnect from the FrameMaker process.

How your client disconnects depends on the situation. With a client that is a plugin for FrameMaker, you can call `F_ApiBailOut()` to terminate the client. After calling `F_ApiBailOut()`, the client's notification points are still registered with the FrameMaker process. If a notification event occurs, the FrameMaker process restarts the client by calling `F_ApiInitialize()` with initialization set to `FA_Init_Subsequent`. When it starts up subsequently, the client's global variable settings are lost.

If the FrameMaker process still exists when your client is completely done communicating with it, your client should call the function `F_ApiDisconnectFromSession()` to break the RPC connection.

Alternatively, the FrameMaker process may have shut down when your client wants to break the connection (for example, due to a user request or due to a command from your client). If so, your client should call the function `F_ApiShutDown()` to close its side of the RPC connection.

Writing a Main routine in Windows.

Windows does not provide a default main routine for remote plugins. You must provide your own main routine. Simply include the following lines in your client:

```
#define DONT_REDEFINE /* We need to use native types. */
#include 'fapi.h'
#include <windows.h>
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR
lpCmdLine, int nCmdShow)
{
return F_ApiRun();
}
```

The routine `F_ApiRun()` is documented in the FDK manuals and is implemented as follows:

```
IntT
F_ApiRun(VoidT)
{
    ConStringT s = F_ApiStartup(NULL);
    if (s)
        F_ApiErr(s);
    else
while (!FA_bailout)
        F_ApiService(NULL);
        F_ApiShutDown();
    return s !=NULL;
}
```

`F_ApiStartup()` and `F_ApiService()` ignore their parameters and should be passed `NULL`.

It is not necessary to call `F_ApiRun()`. You may choose to implement your main routine using these primitives directly. If your program has a windows message loop you need only call `F_ApiStartup(NULL)`.

However if your remote plugin does not call `F_ApiRun()`, it must either periodically check the `FA_bailout` flag or arrange to terminate based on the `FA_NotePostQuitSession` notification. You must make these checks, otherwise `FrameMaker` can terminate leaving your client running.

Compiling and running a sample client

The following code sample is a console application that connects to the default FrameMaker session and gets the name of the active FrameMaker document. Following the code is a lineby-line description of how it works.

```
1. #define DONT_REDEFINE // Console app needs native types
2. #define WBUFLEN 512
3.
4. #include "fdetypes.h"
5. #include "futils.h"
6. #include "fapi.h"
7. #include "fstrings.h"
8. #include <windows.h>
9. #include <ddeml.h> //not required
10. #include <stdarg.h> //not required
11.
12. int main(int argc, char **argv)
13. {
14.     StringT opt_progid;
15.     CLSID pclsid;
16.     LPOLESTR progStr;
17.     HRESULT res;
18.     F_ObjHandleT docId;
19.
20.     // Get the process name.
21.     if(argc == 2)
22.         opt_progid = F_StrCopyString((StringT)argv[1]);
23.     else {
24.         fprintf(stderr, "You must provide a process name.");
25.         return(1);
26.     }
27.
28.     // Convert the process name into a GUID
29.     progStr = (OLECHAR*)malloc( WBUFLEN*sizeof(wchar_t) );
30.     if(0 == MultiByteToWideChar(CP_ACP, 0,
31.         (char *)opt_progid, -1, 31. progStr, WBUFLEN )) {
32.         fprintf(stderr, "failed to allocate\n");
```

```

33.     return(1);
34.     }
35.     if (progStr[0] == '{') // hex-codes within brackets
36.         res = CLSIDFromString(progStr, &pclsid);
37.     else
38.         res = CLSIDFromProgID(progStr, &pclsid);
39.
40.     if(res == S_OK)
41.         F_ApiWinConnectSession(NULL, NULL, &pclsid);
42.     if (!F_ApiAlive()) {
43.         fprintf(stderr, "No connection: %s\n", opt_progid);
44.         return 1;
45.     }
46.     // Print the name of the current document.
47.     docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
48.     if (docId) {
49.         StringT docname = F_ApiGetString(FO_Session, docId,
50.                                         FP_Name);
51.         fprintf(stderr, "Current document: %s\n", docname);
52.         F_ApiDeallocateString(&docname);
53.     } else
54.         fprintf(stderr, "No active document\n");
55.     return 0;
56. }

```

Line 1

To compile this client as a console application, you need to use types that are native to the C language. This statement keeps the FDE from redefining those types.

Lines 20–26

These lines parse the commandline options you pass to the client when you invoke it. You invoke the exe with the name of a FrameMaker process as an argument. To run the default process, use the name FrameMaker.API.1. For example, assuming the exe is named fmRemote.exe, type the following to invoke it with the default FrameMaker process: fmRemote.exe FrameMaker.API.1

For more information, see "[Registering a name for a FrameMaker process](#)".

Lines 28–38

These lines convert the process name into a valid GUID. Note that you need a Unicode string for the process name. The code uses the Win32 API call, `ultiByteToWideChar()` to convert the process name to Unicode. It then uses the Win32 API functions `CLSIDFromProgID()` or `CLSIDFromString()` to get the GUID for the specified process.

Lines 40–45

If you successfully retrieve a GUID for the process, these lines make the connection to a FrameMaker session.

Lines 46–56

Now that the client has connected with a session, it can use the FDK to interact with that session. These lines are standard FDK code to get the name of the active document for the current session. You can add code to perform other actions such as adding menus to the application window, manipulating the active document, or anything else you can do via the FDK.

.....
IMPORTANT: *Because they don't have a Windows message processing loop, console applications cannot handle notifications from the FDK, such as menu commands or notifications such as `FA_Note_PreSaveDoc`.*
.....

Compiling and registering the sample client

To compile the sample client in Microsoft Visual Studio 2010, follow these steps:

1. Create a project for a console application.

Use the Project Wizard to create a new project for a console application.

2. Set up the project options and settings as described in "[Compiling, Registering, and Running FDK Clients](#)".

.....
IMPORTANT: *Your link settings must include `fdk.lib` and `api.lib`, but neither `fndbms32.lib` nor `fmdebug.lib`. In previous versions of the FDK, `fndbms32.lib` and `fmdebug.lib` were required to compile. These libraries are now obsolete, but we include them so you don't have to change the link settings to compile existing FDK projects. If a remote client fails to start up and you see these libraries mentioned in the error text, then you must remove them from your link settings and recompile.*
.....

Compile the client.

4. Register the client

There are three ways to register an asynchronous client. See "[Registering asynchronous clients](#)".

You must also be sure the end user has a correct installation to run asynchronous clients. See "[End user installations](#)".

5. Connect the client with a named FrameMaker process.

To connect with a named FrameMaker process:

- On your machine, register the FrameMaker process as a server. See "[Registering multiple FrameMaker processes as servers](#)". Be sure to register it with a name. See "[Registering a name for a FrameMaker process](#)".
- In a command window, type the filename for the client, followed with the name of the FrameMaker process the argument.
- To connect to the default FrameMaker process, use the process name, FrameMaker.API.1.

For example, type `remote.exe process_name`, where `process_name` is the name you assigned to a FrameMaker process. Note that unless you registered the process to start up automatically, that process must be running when you invoke the sample client.

Summary of supporting functionality

To support communication with a FrameMaker process, the FDK provides the following functions:

Function	Purpose
<code>F_ApiWinConnectSession()</code>	Initiates communication between the calling process and an identified FrameMaker process
<code>F_ApiDisconnectFromSession()</code>	Severs communication with a FrameMaker process
<code>F_ApiSetClientDir()</code>	Identifies a directory the FrameMaker process associates with an unregistered client
<code>F_ApiShutDown()</code>	Closes a client's connection with the API
<code>F_ApiWinInstallDefaultMessageFilter()</code>	Registers the default FDK message filter for a COM session.
<code>F_ApiService()</code>	useful if you are providing a replacement for <code>F_ApiRun()</code> .
<code>F_ApiStartup()</code>	See the description after the table

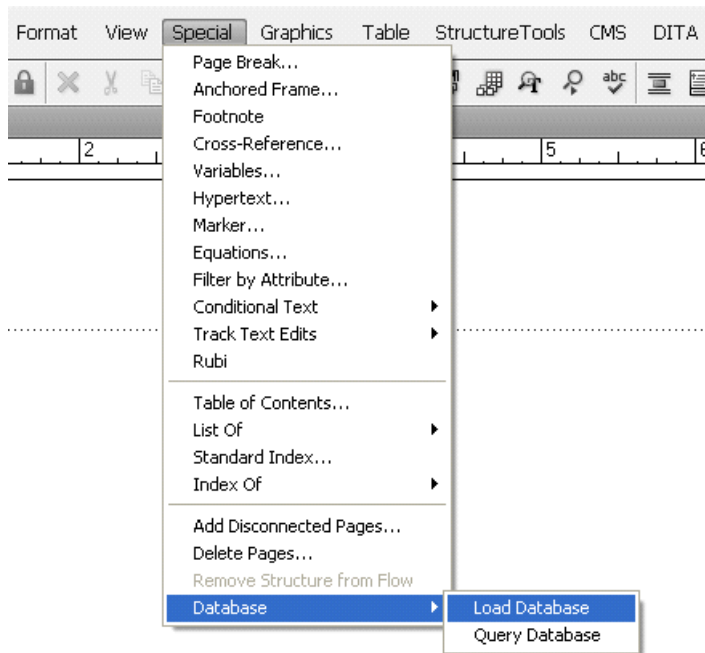
Function	Purpose
<code>F_ApiAlive()</code>	
<code>F_ApiErr(message)</code>	Prints client name and message to console.
<code>F_ApiRun</code>	provides the minimum functionality required in an FDK client's <code>main()</code> function

Using `F_ApiStartup(F_FdFuncT)` the `F_FdFuncT` argument is ignored because Windows RPC is not based on sockets. `F_ApiStartup` queries the application's version information for client configuration data, if present, and connects to FrameMaker.

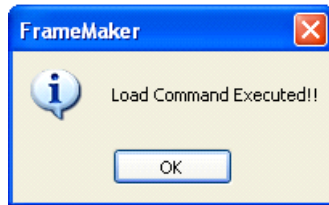
For information on these functions and properties, see the *FDK Programmer's Reference* guide.

Example: adding menus and commands

The following code adds a menu named "Database" to the Special menu. The menu has two commands "Load Database" and "Query Database".



When the user selects either of the commands, a prompt is displayed.



```
#include "fapi.h"

#define LOAD 1
#define QUERY 2

VoidT F_ApiInitialize(initialization)
IntT initialization;
{
F_ObjHandleT specialMenuId, databaseMenuId;
/* Get the ID of the special menu. */
specialMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
"SpecialMenu");
/* Define the menu and add it to the Edit menu. */
databaseMenuId = F_ApiDefineAndAddMenu(specialMenuId,
"DatabaseMenu",
"Database");
/* Define the commands and add them to the Special menu. */
F_ApiDefineAndAddCommand(LOAD, databaseMenuId,
"LoadDatabase", "Load Database", "\\!LD");
F_ApiDefineAndAddCommand(QUERY, databaseMenuId,
"QueryDatabase", "Query Database", "\\!QD");
}
VoidT F_ApiCommand(command)
IntT command;
{
switch(command)
{
case LOAD: /* Code to load database goes here. */
F_ApiAlert((ConStringT)"Load Command Executed!!\n" ,
```

```
FF_ALERT_CONTINUE_NOTE);  
break;  
case QUERY: /* Code to query database goes here. */  
    F_ApiAlert((ConStringT)"Query Command Executed!!\n" ,  
FF_ALERT_CONTINUE_NOTE);  
break;  
}  
}
```

Next Steps

By now, you would have become familiar with the basic operations of the FDK. Here are the suggested next steps that will help you use the FDK more effectively:

- Study the Programmer's Guide to understand the detailed flow and usage model
- Review the samples provided (available in the `samples` folder of the FDK installation) and write your own program modeled on them.
- Refer to the *FDK Programmer's Reference* for details of syntax and examples

PART II



Frame Product Architecture

Frame Session Architecture

:
:
:
:

This chapter discusses the general organization of FrameMaker product sessions from a programmer's perspective. It provides useful background information for programmers who want to use the Frame API.

Identifying objects

The API assigns a unique ID to each object. Most API functions that manipulate objects require you to specify this ID. An object's ID is valid only as long as the object is available in the current FrameMaker product session. For example, suppose you have a document with a rectangle drawn in it. When you open the document, the API assigns an ID to the `FO_Rectangle` object that represents the rectangle. As long as the document is open, the ID of the `FO_Rectangle` object remains the same. However, if you exit the document and then reopen it, the API may assign a new ID to the rectangle.

In addition to IDs, there are two types of identifiers that are *persistent* between sessions:

- Unique object names
- Unique persistent identifiers (UIDs)

Each object generally has either an `FP_Name` property specifying a unique object name or an `FP_Unique` property specifying a UID.

Unique object names

There are many types of objects that you can assign unique names to in the user interface. These objects, which are called *named objects*, include:

- `FO_Book`
- `FO_CharFmt`
- `FO_Color`
- `FO_CombinedFontDfn`
- `FO_Command`
- `FO_CondFmt`

- `FO_ElementDef`
- `FO_FmtChangeList`
- `FO_MarkerType`
- `FO_MasterPage`
- `FO_Menu`
- `FO_MenuItemSeparator`
- `FO_PgfFmt`
- `FO_RefPage`
- `FO_RulingFmt`
- `FO_TiFlow`
- `FO_TiText`
- `FO_TblFmt`
- `FO_UnanchoredFrame` (named frames on reference pages only)
- `FO_VarFmt`
- `FO_XRefFmt`

The API provides a function named `F_ApiGetNamedObject()`, which gets the ID of a named object with a specified name.

.....

IMPORTANT: *A document can contain several flows with the same name. For example, a document can contain several A flows. To get the ID of a specific flow, first get the ID of a text frame in that flow, for example, the text frame for the current text location, and then query the text frame's `FP_Flow` property.*

.....

Unique persistent identifiers (UIDs)

The API and MIF identify unnamed objects with UIDs. An unnamed object is an object that doesn't have a unique name. For example, `FO_Pgf` objects are unnamed.

UIDs are unique within documents. An object's UID remains the same as long as the object is in the same document. The API provides a function named `F_ApiGetUniqueObject()`, which gets an object's ID from its UID.

.....

IMPORTANT: *If you copy an object and then paste it, the FrameMaker product considers the pasted object a new object and assigns a new UID to it. This is also true for a paragraph that is conditionalized. If the entire paragraph is of a given*

condition, and that condition is hidden and then shown, the paragraph will have a new UID.

.....

Representing object characteristics with properties

Each object has a *property list*, or set of properties that represent its characteristics. Each property has a *value* associated with it. For example, if a paragraph has two tabs, the value of its `FP_NumTabs` property is `2`. A property value can be more than an integer. It can also be a string, a pointer to a structure that contains a set of strings, or a variety of other things. The following table summarizes the different data types property values can be.

Property data type	What the property value represents
<code>IntT</code>	An integer, enum, boolean, or ordinal value. For many <code>IntT</code> properties, the API provides defined constants, such as <code>True</code> and <code>False</code> .
<code>F_IntsT</code>	A set of integers or a set of IDs.
<code>F_UIntsT</code>	A set of unsigned integers.
<code>MetricT</code>	A measurement value.
<code>F_MetricsT</code>	A set of metrics.
<code>StringT</code>	A character string.
<code>F_StringsT</code>	A set of character strings.
<code>F_ObjHandleT</code>	The ID of another object.
<code>F_PointsT</code>	A set of x-y coordinate pairs.
<code>F_TabsT</code>	A set of tab descriptions.
<code>F_TextLocT</code>	A point (location) in text.
<code>F_TextRangeT</code>	A range or selection of text.
<code>F_ElementCatalogEntriesT</code>	The list of elements in the Element Catalog.
<code>F_AttributeDefsT</code>	An set of attribute definitions.
<code>F_AttributesT</code>	An set of attributes.
<code>F_ElementRangeT</code>	An element selection.

The API uses `MetricT` data to express measurement values. This manual uses constants to represent conventional measurement system units as `MetricT` data. For example, the constant `in` represents an inch and the constant `pts` represents a point in `MetricT` units, for example 5 inches ($5 * 4718592$) are represented as `5*in`.

For more information on the `MetricT` type and other data types and data structures listed in the table above, see chapter, “Data Types and Structures Reference,” of the FDK Programmer’s Reference.

Many property values are pointers to data structures. For example, `FO_Doc` objects have a property named `FP_Dictionary` that specifies words that the Spelling Checker will permit in a document. `FP_Dictionary` is an `FT_Strings` property. Its value is a pointer to an `F_StringsT` structure, which is defined as:

```
typedef struct {
    UIntT len;      /* Number of permitted words */
    StringT *val;   /* Vector of permitted words */
} F_StringsT;
```

Property lists

At the highest level, the API represents each object’s property list with a `F_PropValsT` structure. The `F_PropValsT` structure is defined as:

```
typedef struct {
    UIntT len;      /* Number of properties in list */
    F_PropValT *val; /* Property-value pairs */
} F_PropValsT;
```

The `F_PropValT` structure, which provides an individual property-value pair, is defined as:

```
typedef struct {
    F_PropIdentT propIdent; /* The property identifier */
    F_TypedValT propVal;   /* The property value */
} F_PropValT;
```

The `F_PropIdentT` structure, which identifies a property by either its property number constant (one of the constants beginning with `FP_`) or a property name, is defined as:

```
typedef struct {
    IntT num; /* The property number constant */
    StringT name; /* The property name */
} F_PropIdentT;
```

Most properties are identified by property number constants. Only inset facets, a special type of properties, are identified by names. For information on insets, see Chapter 12,

“Using Imported Files and Insets.” If a property is identified by a name, `F_PropIdentT.num` is 0.

The `F_TypedValT` structure is defined as:

```
typedef struct {
    IntT valType; /* The type of value. See table below */
    union {
        StringT sval; /* String value */
        F_StringsT ssva; /* Set of strings */
        F_MetricsT msval; /* Set of metrics */
        F_PointsT psval; /* Set of points */
        F_TabsT tsval; /* Set of tabs */
        F_TextLocT tlval; /* Text location */
        F_TextRangeT trval; /* Text range */
        F_ElementCatalogEntriesT csval; /* Element Catalog */
        F_AttributeDefsT adsva; /* Attribute definitions */
        F_AttributesT asval; /* Attribute values */
        F_ElementRangeT *erng; /* Element range */
        F_IntsT isval; /* Set of integers */
        F_UIntsT uisval; /* Set of unsigned integers */
        IntT ival; /* Integer */
    } u;
} F_TypedValT;
```

The constants used in the `valType` field are described in the following table.

valType constant	Property data type	u field
<code>FT_Integer</code>	<code>IntT</code>	<code>ival</code>
<code>FT_Ints</code>	<code>F_IntsT</code>	<code>isval</code>
<code>FT_Metric</code>	<code>MetricT</code>	<code>ival</code>
<code>FT_Metrics</code>	<code>F_MetricsT</code>	<code>msval</code>
<code>FT_String</code>	<code>StringT</code>	<code>sval</code>
<code>FT_Strings</code>	<code>F_StringsT</code>	<code>ssval</code>
<code>FT_Id</code>	<code>F_ObjHandleT</code>	<code>ival</code>
<code>FT_Points</code>	<code>F_PointsT</code>	<code>psval</code>
<code>FT_Tabs</code>	<code>F_TabsT</code>	<code>tsval</code>
<code>FT_TextLoc</code>	<code>F_TextLocT</code>	<code>tlval</code>
<code>FT_TextRange</code>	<code>F_TextRangeT</code>	<code>trval</code>

valType constant	Property data type	u field
FT_UInts	F_UIntsT	uisval
FT_UBytes	F_UBytesT	No field
FT_ElementCatalog	F_ElementCatalogEntriesT	csval
FT_AttributeDefs	F_AttributeDefsT	adsval
FT_Attributes	F_AttributesT	asval
FT_ElementRange	F_ElementRangeT	erng

.....
IMPORTANT: *Integer (IntT), metric (MetricT), and ID (F_ObjHandleT) values are all put in the ival field of the u union.*

Example

Suppose the user creates a paragraph format named Heading, which has a 1-inch left indent and has Keep With Next Paragraph turned on. The API represents this paragraph format with an `FO_PgfFmt` object. The following are some of the object's properties and their values.

Property	Data type	Value
FP_Name	StringT	Heading
FP_KeepWithNext	IntT	True
FP_LeftIndent	MetricT	1*in

The property list for the `FO_PgfFmt` object and the properties in the previous table are represented graphically in Figure 1-1. `FO_PgfFmt` objects have many other properties that are not shown in the illustration.

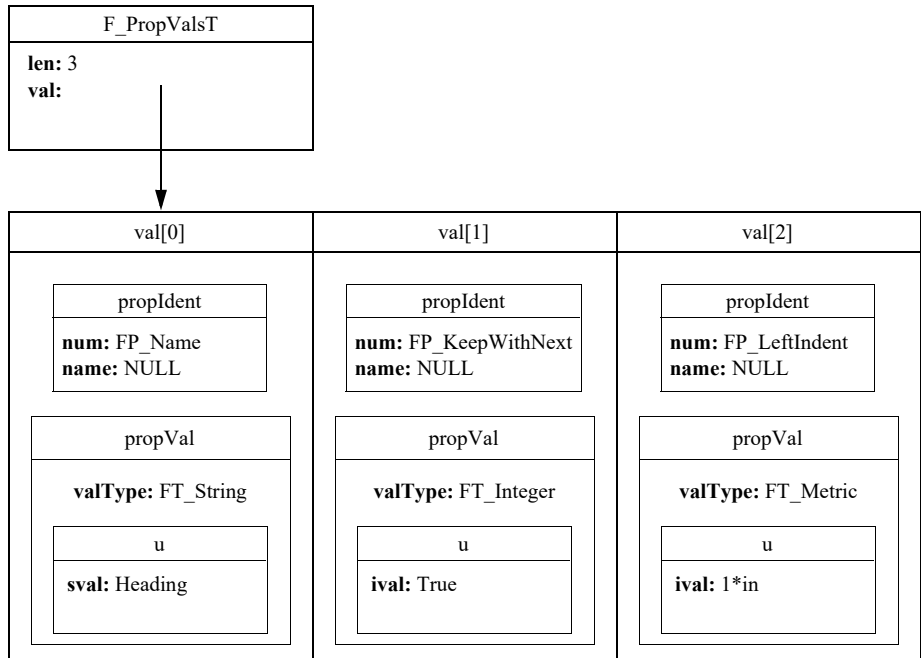


Figure 1-1 Some `FO_PgfFmt` properties

FrameMaker product sessions

The fundamental entity in Frame architecture is a session. Each instance of a FrameMaker product that the user starts is a session. FrameMaker allows the user to have many open documents and books in a session.

Of the open documents and books in a session, only one document or book is active at a time. An open document or book is active if it has the input focus.

How the API represents sessions

The API represents each FrameMaker product session with an `FO_Session` object, whose properties provide the following categories of information about the session:

- System information, such as the operating system, the current FrameMaker product version, and the current directory
- The automatic save settings
- Names of fonts available on the system
- IDs of the objects that represent open and active documents and books
- Whether the FrameMaker product reformats and redisplay documents after changes have been made
- Whether element reformatting and validation is turned on (for FrameMaker structured documents)

Suppose you start FrameMaker on a Window System platform and open a document named `mydoc`. The API represents this session with an `FO_Session` object. The following are some of its properties.

Property	Type	Value
<code>FP_ProductName</code>	<code>FT_String</code>	FrameMaker
<code>FP_VersionMajor</code>	<code>FT_Integer</code>	5
<code>FP_WindowSystem</code>	<code>FT_String</code>	Windows
<code>FP_AutoSaveSeconds</code>	<code>FT_Integer</code>	300
<code>FP_ActiveDoc</code>	<code>FT_Id</code>	ID of the object that represents <code>mydoc</code>

How the API indicates which documents and books are open

The API represents a document with an `FO_Doc` object. The API maintains a linked list of the `FO_Doc` objects that represent a session's open documents. The `FO_Session` property, `FP_FirstOpenDoc`, specifies the ID of the first `FO_Doc` object in the list. The `FO_Doc` property, `FP_NextOpenDocInSession`, specifies the ID of the next `FO_Doc` object in the list. The list of `FO_Doc` objects that represent open documents is not in any particular order. The `FO_Doc` object specified by `FP_FirstOpenDoc` does *not* necessarily represent the first document the user opened.

The API represents a book with an `FO_Book` object. The API also maintains the `FO_Book` objects that represent the session's open books in a linked list. The `FO_Session` property, `FP_FirstOpenBook`, specifies the ID of the first `FO_Book` object in the list. The `FO_Book` property, `FP_NextOpenBookInSession`, specifies the next `FO_Book` object in the list. As with the list of `FO_Doc` objects, the linked list of `FO_Book` objects is not in any particular order.

How the API indicates which document or book is active

`FO_Session` has two properties, `FP_ActiveDoc` and `FP_ActiveBook`, that specify the IDs of the objects that represent the active document and the active book. Only one document or one book can be active at a time. If there is no active document or book, these properties are set to 0. Invisible documents and books can't be active.

Example

Suppose you start FrameMaker and open the books and documents shown in Figure 1-2. The Frame API represents the session with the objects shown in Figure 1-3.

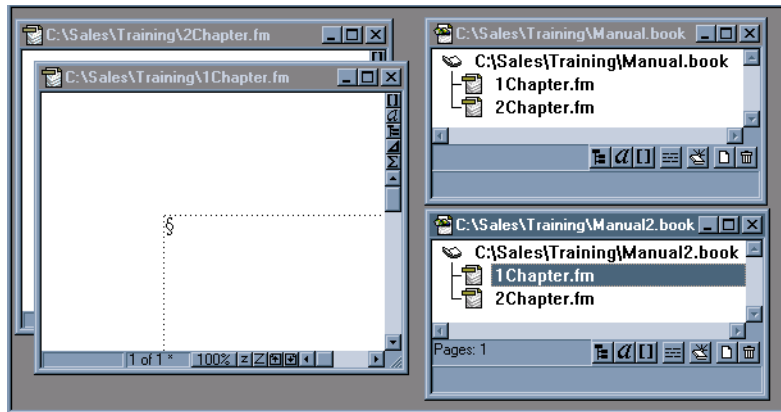


Figure 1-2 A FrameMaker session with open documents and books

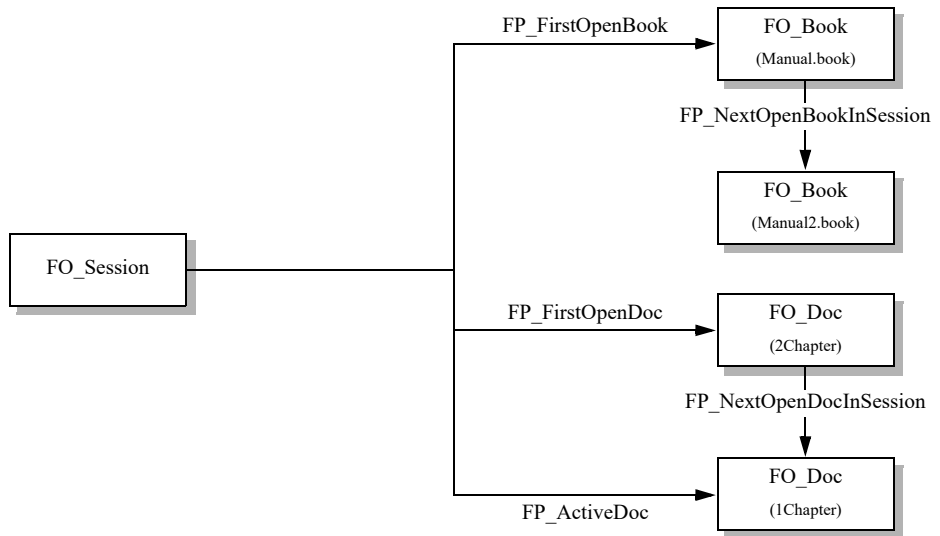


Figure 1-3 API representation of a session with open documents and books

Although `Manual.book` is iconified, the API still considers it open. Although `FP_FirstOpenDoc` specifies `2Chapter`, it is not necessarily the first document that was opened.

How the API indicates which fonts are available in a session

The following `FO_Session` properties specify which fonts are available in the current session:

- `FP_FontFamilyNames` specifies the available families, for example, `Helvetica` and `Times`.
- `FP_FontVariationNames` specifies the available variations, for example, `Narrow` and `Oblique`.
- `FP_FontWeightNames` specifies the available weights, for example, `Bold` and `Regular`.
- `FP_FontAngleNames` specifies the available angles, for example, `Italic` and `Regular`.

The `FP_FontFamilyNames`, `FP_FontVariationNames`, `FP_FontWeightNames`, and `FP_FontAngleNames` properties determine which choices appear in the Family, Weight, Angle, and Variation fields of the Character Designer and Paragraph Designer, and the pull-right menu items in the Format menu. Each of the properties specifies an `F_StringsT` structure, which is defined as:

```
typedef struct {  
    UIntT len; /* Number of strings */  
    StringT *val; /* Font names */  
} F_StringsT;
```

For example, if Courier, Helvetica, and Times are the only font families available in the current session, the fields of the `F_StringsT` structure specified by `FP_FontFamilyNames` have the following values:

```
len: 4  
val: { "<Reserved>", "Courier", "Helvetica", "Times" }
```

Properties that specify font families, angles, weights, and variations use the index of the `val` array. For example, the `FO_CharFmt` property, `FP_FontFamily`, specifies the font family for a character format. Given the `F_StringsT` values shown above, if the font family for a character format is Helvetica, the value of the format's `FP_FontFamily` property is 2.

Although a specific angle, weight, or variation may be in one of the lists described above, it may not be available for all combinations of font families, angles, weights, and variations. For example, the Bold weight may be available for Times and Helvetica, but not for Zapf Chancery. The FDK provides a convenience function named `F_ApiFamilyFonts()`, which returns all the permutations of font families, angles, weights, and variations in a FrameMaker product session. For more information, see “`F_ApiFamilyFonts()`” in the FDK Programmer’s Reference guide.

Frame Document Architecture

.....

⋮

This chapter describes Frame documents and their components and shows how the Frame API represents them.

Documents

A document is a set of pages with graphic objects and text that the user creates with a FrameMaker product and stores in a file.

What the user sees

When you create a new document, you can use a template to create it, or you can create a custom document. Any document can be a template. Because the FrameMaker product copies everything from a template to a new document, most users prefer to use templates containing only layout and formatting information. FrameMaker provides ready-made templates for a variety of document types.

You can't create a document completely from scratch—the document must have a certain set of default objects for the FrameMaker product to work correctly. To ensure that all documents have this set of objects, the FrameMaker product always uses a template to create a new document. Even if you choose the Custom document option, the FrameMaker product creates the new document from a default template. This custom document template is specified in the `maker.ini` file.

When the user attempts to create a new document from an ASCII text file or a MIF file that doesn't provide the necessary objects, FrameMaker uses the ASCII template specified in the `maker.ini` file.

When you instruct FrameMaker to save a document, it lists the document's objects and their properties in a file. By default, FrameMaker writes the information to a Frame binary format file. You can also choose to save a document as a MIF file.

How the API represents documents

A document actually consists of much more than text and graphic objects. It includes information specifying a variety of other things, such as formatting, user preferences, and the FrameMaker product's default behavior. The API represents the information in a document with a set of objects. The following table summarizes the information a document can contain and the objects the API uses to represent it.

Type of information	Function	Types of objects that represent it
Global document information	Specifies the document's general characteristics, some aspects of the FrameMaker product's behavior when the document has input focus, and IDs of other objects that constitute the document	FO_Doc
Pages	Organize text and graphic objects in the document	FO_BodyPage FO_MasterPage FO_RefPage FO_HiddenPage
Graphic objects	Describe graphic objects in the document	FO_UnanchoredFrame FO_AFrame FO_Group FO_Arc FO_Rectangle FO_Ellipse FO_RoundRect FO_Polyline FO_Polygon FO_Line FO_TextLine FO_TextFrame FO_Inset FO_Math
Text columns	Contain text	FO_SubCol
Text frames	Contain text	FO_TextFrame

Type of information	Function	Types of objects that represent it
Text flows	Specify how text frames in the document are linked	FO_Flow
Paragraph Catalog formats	Specify tags that the user can apply to a paragraph to change its formatting	FO_PgFfmt
Paragraphs	Contain the document's text and provide formatting information for individual paragraphs	FO_Pgf
Character Catalog formats	Specify tags that the user can apply to a selection of characters to change its formatting	FO_CharFfmt
Condition formats	Specify tags that the user can apply to text to indicate that it belongs to a particular variation of the document	FO_CondFfmt
Markers	Describe placeholders that contain hidden text	FO_Marker
Marker types	Specifies a named category of markers	FO_MarkerType
Cross-reference formats	Specify the wording and typographic style of cross-references	FO_XRefFfmt
Cross-reference instances	Describe instances of cross-references in the document	FO_XRef
Variable formats	Specify units of text and system-supplied information that the user can use multiple times in a document	FO_VarFfmt
Variable instances	Describe instances of variables in the document	FO_Var
Footnotes	Describe footnotes	FO_Fn
Table ruling formats	Specify rulings and shadings that the user can apply to individual table cells	FO_RulingFfmt
Table Catalog formats	Specify table formats that the user can apply to a table and that provide default numbers of columns and rows for new tables	FO_TblFfmt
Tables	Describe instances of tables in the document and specify formatting information, such as alignment, ruling, and shading	FO_Tbl FO_Row FO_Cell
Colors	Specify colors that the user can apply to graphic objects and text	FO_Color

Type of information	Function	Types of objects that represent it
Text insets	Describe text that is imported by reference	FO_TiApiClient FO_TiFlow FO_TiText FO_TiTextTable
Structural element definitions	Specify tags that specify the organization of parts of a structured document	FO_ElementDef
Structural element instances	Describe instances of structural elements in a structured document	FO_Element
Format rules	Specify sets of format rule clauses	FO_FmtRule
Format rule clauses	Specify which formats to apply to elements in various contexts	FO_FmtRuleClause
Format change list	Specify format changes applied to an element in a specific context	FO_FmtChangeList
Rubi composites	Describe the oyamoji (base word) and rubi (phonetic spelling) of certain words in Asian text	FO_Rubi
Combined font definitions	Describe pairs of Asian and Western fonts that are treated as a single font family	FO_CombinedFontDfn

The other sections of this chapter discuss the different types of information in a document.

How the API organizes the objects that constitute a document

The API uses an `FO_Doc` object to organize the objects that constitute a document. `FO_Doc` objects have a number of properties that specify the IDs of other objects in the document. Many of these properties specify the ID of the first object in a linked list of objects. For example, `FP_FirstPgFfmtInDoc` specifies the first `FO_PgfFfmt` object (Paragraph Catalog format) in the list of `FO_PgfFfmt` objects in the document. Each `FO_PgfFfmt` object has a `FP_NextPgFfmtInDoc` property that specifies the next `FO_PgfFfmt` object in the list. If you want to get all the `FO_PgfFfmt` objects in a document, you get the `FO_PgfFfmt` object specified by `FP_FirstPgFfmtInDoc` and traverse the links to the other objects.

Document object property	Object that the property specifies
<code>FP_FirstGraphicInDoc</code>	The first graphic object (for example, <code>FO_UnanchoredFrame</code> or <code>FO_Line</code>) in the list of graphic objects
<code>FP_FirstColorInDoc</code>	The first <code>FO_Color</code> in the list of <code>FO_Color</code> objects
<code>FP_FirstPgfInDoc</code>	The first <code>FO_Pgf</code> in the list of <code>FO_Pgf</code> objects
<code>FP_FirstMarkerInDoc</code>	The first <code>FO_Marker</code> in the list of <code>FO_Marker</code> objects
<code>FP_FirstMarkerTypeInDoc</code>	The first <code>FO_MarkerType</code> , in the list of marker types
<code>FP_FirstVarInDoc</code>	The first <code>FO_Var</code> in the list of <code>FO_Var</code> objects
<code>FP_FirstVarFmtInDoc</code>	The first <code>FO_VarFmt</code> in the list of <code>FO_VarFmt</code> objects
<code>FP_FirstXRefInDoc</code>	The first <code>FO_XRef</code> in the list of <code>FO_XRef</code> objects
<code>FP_FirstXRefFmtInDoc</code>	The first <code>FO_XRefFmt</code> in the list of <code>FO_XRefFmt</code> objects
<code>FP_FirstFnInDoc</code>	The first <code>FO_Fn</code> in the list of <code>FO_Fn</code> objects
<code>FP_FirstTblInDoc</code>	The first <code>FO_Tbl</code> in the list of <code>FO_Tbl</code> objects
<code>FP_FirstFlowInDoc</code>	The first <code>FO_Flow</code> in the list of <code>FO_Flow</code> objects
<code>FP_FirstPgfFfmtInDoc</code>	The first <code>FO_PgfFfmt</code> in the list of <code>FO_PgfFfmt</code> objects
<code>FP_FirstCharFmtInDoc</code>	The first <code>FO_CharFmt</code> in the list of <code>FO_CharFmt</code> objects

Document object property	Object that the property specifies
FP_FirstCondFmtInDoc	The first <code>FO_CondFmt</code> in the list of <code>FO_CondFmt</code> objects
FP_FirstTblFmtInDoc	The first <code>FO_TblFmt</code> in the list of <code>FO_TblFmt</code> objects
FP_FirstRulingFmtInDoc	The first <code>FO_RulingFmt</code> in the list of <code>FO_RulingFmt</code> objects
FP_FirstSelectedGraphicInDoc	The first graphic object in the list of selected graphic objects
FP_MainFlowInDoc	<code>FO_Flow</code> that represents the main flow
FP_FirstElementDefInDoc	First structural element definition in the list of element definitions in a FrameMaker document
FP_FirstFmtChangeListInDoc	First format change list in the list of format change lists in a document
FP_FirstBodyPageInDoc or FP_LastBodyPageInDoc	The first or last <code>FO_BodyPage</code> in the list of <code>FO_BodyPage</code> objects
FP_FirstMasterPageInDoc or FP_LastMasterPageInDoc	The first or last <code>FO_MasterPage</code> in the list of <code>FO_MasterPage</code> objects
FP_FirstRefPageInDoc or FP_LastRefPageInDoc	The first or last <code>FO_RefPage</code> in the list of <code>FO_RefPage</code> objects
FP_HiddenPage	The hidden page (<code>FO_HiddenPage</code>)
FP_SelectedTbl	The selected table object
FP_FirstTiInDoc	The first <code>FO_TiApiClient</code> , <code>FO_TiFlow</code> , <code>FO_TiText</code> , or <code>FO_TiTextTable</code> in the list of text insets
FP_FirstRubiInDoc	The first <code>FO_Rubi</code> in the list of rubi composites
FP_FirstCombinedFontDfnInDoc	The first <code>FO_CombinedFontDfn</code> in the list of combined font definitions

`FP_FirstBodyPageInDoc`, `FP_FirstMasterPageInDoc`, and `FP_FirstRefPageInDoc` point to the lists of pages in a document. These lists are ordered to reflect the order of the pages. All other lists (including the list of `FO_Pgf` objects) are not ordered. The terms *first* and *last* indicate only the position of the objects in an arbitrarily ordered list. There is no guarantee that a more recently added object will come later in a list, nor is there a guarantee that the order of a list will remain the same as the document is modified.

Global document information

FrameMaker products allow you to set *global document information*, characteristics that apply generally to an entire document.

What the user sees

Global document information includes the following formatting characteristics:

- Document page properties, which specify the document's page numbering and pagination style
- Document condition properties, which specify whether conditional text appears and whether formatting associated with condition tags overrides other formatting
- Document and table footnote properties, which specify the appearance of the footnotes, such as the footnote numbering and the default paragraph format
- Change bar properties, which specify the appearance and position of change bars in the document
- The current text selection or insertion point

There is also global document information that affects how the FrameMaker product behaves when the document is active. This type of global information includes:

- The document dictionary, which lists words that you want the FrameMaker product Spelling Checker to ignore
- Type-in properties, which specify whether Smart Spaces or Smart Quotes is enabled
- Equation properties, which specify default symbol sizes and fonts the FrameMaker product uses when you add equations to the document
- Printing properties, which specify the defaults that appear in the Print dialog box, such as the printer name and the range of pages to print
- View properties, which specify how FrameMaker displays and scrolls the document in the window
- Structure properties, which specify whether element boundaries appear and how the Element Catalog appears for a structured document in a session.

FrameMaker saves most of the global document information with each document. For example, if you set the zoom for a document to 140 percent and save and exit a document, the next time you open the document, the zoom will be set to 140 percent.

How the API represents global document information

The Frame API represents global document information with `FO_Doc` object properties.

How the API represents the selection in a document

The API uses several properties to specify what is selected in a document:

- `FP_TextSelection` specifies a structure that provides the location of the insertion point or the beginning and end of a text selection.
- `FP_FirstSelectedGraphicInDoc` specifies the ID of the first graphic in the list of selected graphics in a document.
- `FP_SelectedTbl` specifies the ID of a table that contains the insertion point or some selected cells.
- `FP_ElementSelection` specifies the range of elements selected if the document is a structured document in a session.

The following table summarizes the different types of selection in an unstructured document and how these properties are set to represent them.

Selection state	How selection properties are set
No object is selected. There is no text selection or insertion point.	<code>FP_TextSelection</code> specifies an <code>F_TextRangeT</code> structure for which the <code>objId</code> and <code>offset</code> fields of <code>F_TextRangeT.beg</code> and <code>F_TextRangeT.end</code> are set to 0. <code>FP_FirstSelectedGraphicInDoc</code> is 0. <code>FP_SelectedTbl</code> is 0.
One or more graphic objects are selected.	<code>FP_TextSelection</code> specifies an <code>F_TextRangeT</code> structure for which the <code>objId</code> and <code>offset</code> fields of <code>F_TextRangeT.beg</code> and <code>F_TextRangeT.end</code> are set to 0. <code>FP_FirstSelectedGraphicInDoc</code> specifies the ID of the first selected graphic in the document's list of selected graphics. <code>FP_SelectedTbl</code> is 0.
There is an insertion point or text selection (that isn't in a table cell, but may include table anchors).	<code>FP_TextSelection</code> specifies the location of the text selection or insertion point <code>FP_FirstSelectedGraphicInDoc</code> is 0. <code>FP_SelectedTbl</code> is 0.

Selection state	How selection properties are set
<p>There is an insertion point or text selection <i>within</i> a single table cell.^a</p>	<p><code>FP_TextSelection</code> specifies the location of the text selection or insertion point within the cell; for example, the ID of the paragraph containing the insertion point, and the offset within that paragraph.</p> <p><code>FP_FirstSelectedGraphicInDoc</code> is 0.</p> <p><code>FP_SelectedTbl</code> specifies the ID of the table containing the cell.</p> <p>If the current selection is in a paragraph, the paragraph's <code>FP_InTextObj</code> property specifies the ID of the cell that contains the selection. The cell's <code>FP_CellColNum</code> property specifies the column number, and the cell's <code>FP_CellRow</code> property specifies the ID of its row.</p>
<p>An entire cell or set of cells is selected.</p>	<p><code>FP_TextSelection</code> specifies an <code>F_TextRangeT</code> structure for which the <code>objId</code> and <code>offset</code> fields of <code>F_TextRangeT.beg</code> and <code>F_TextRangeT.end</code> are set to 0.</p> <p><code>FP_FirstSelectedGraphicInDoc</code> is 0.</p> <p><code>FP_SelectedTbl</code> specifies the ID of the table containing the cell. The table properties <code>FP_TopRowSelection</code> and <code>FP_BottomRowSelection</code> specify the IDs of the first and last rows containing selected cells. The <code>FP_LeftColNum</code> and <code>FP_RightColNum</code> properties of the table specify the index numbers of the outermost columns in the selection.</p>

a. If an entire cell is selected, there is no text selection.

How the API represents the element selection in a structured FrameMaker document

In a structured FrameMaker document, the selection properties described in the previous section behave as they would in an unstructured document. However, structured FrameMaker documents have an additional selection property, `FP_ElementSelection`, which specifies the selection in terms of the selected element range or `F_ElementRangeT` structure.

The `F_ElementRangeT` structure is defined as:

```
typedef struct {
    F_ElementLocT beg; /* Beginning of the element range. */
    F_ElementLocT end; /* End of the element range. */
} F_ElementRangeT;
```

The `F_ElementLocT` structure specifies a location within an element. It is defined as:

```
typedef struct {
    F_ObjHandleT parentId; /* Parent element ID. */
    F_ObjHandleT childId; /* Child element ID. */
    IntT offset; /* Offset within child/parent element. */
} F_ElementLocT;
```

The following table summarizes the different types of selection in a structured FrameMaker document and how the fields of the `F_ElementRangeT` structure specified by the `FP_ElementSelection` property are set to represent them.

Selection state	What the fields of the <code>F_ElementRangeT</code> structure specify
No object is selected. There is no text selection or insertion point.	<code>beg.parentId: 0</code> <code>beg.childId: 0</code> <code>beg.offset: 0</code>
One or more graphic objects are selected.	<code>end.parentId: 0</code> <code>end.childId: 0</code> <code>end.offset: 0</code>
There is an insertion point or text selection within an element that has no subelements.	<code>beg.parentId: ID of the element containing the insertion point or selection</code> <code>beg.childId: ID of the child element immediately following the insertion point or the beginning of the text selection</code> <code>beg.offset: offset of the beginning of the selection or insertion point from the beginning of the element containing it</code> <code>end.parentId: ID of the element containing the insertion point or selection</code> <code>end.childId: ID of the child element immediately following the insertion point or the end of the text selection</code> <code>end.offset: offset of the end of the selection or insertion point from the beginning of the element containing it</code>

Selection state	What the fields of the F_ElementRangeT structure specify
An entire element or range of elements (excluding the highest level element) is selected.	beg.parentId: ID of the element containing the first selected element beg.childId: ID of the first selected element beg.offset: 0 end.parentId: ID of the element containing the first selected element end.childId: ID of the sibling element following the last selected element, or 0 if there is no sibling element following the last selected element end.offset: 0
The highest level element is selected.	beg.parentId: 0 beg.childId: ID of the highest-level element beg.offset: 0 end.parentId: 0 end.childId: 0 end.offset: 0

Example

Suppose you create the document shown in Figure 2-1.

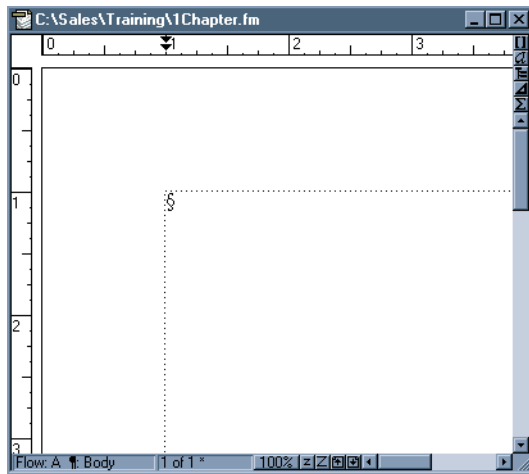


Figure 2-1 A document

The API represents the document with an `FO_Doc` object. The following table lists some of its properties.

Property	Type	Value
<code>FP_Name</code>	<code>StringT</code>	<code>C:\Sales\Training\1Chapter</code>
<code>FP_ViewBorders</code>	<code>IntT</code>	<code>True</code>
<code>FP_ViewRulers</code>	<code>IntT</code>	<code>True</code>
<code>FP_ViewPageScrolling</code>	<code>IntT</code>	<code>FV_SCROLL_VERTICAL</code>
<code>FP_Zoom</code>	<code>MetricT</code>	<code>1 << 16</code>
<code>FP_IsIconified</code>	<code>IntT</code>	<code>False</code>
<code>FP_ViewTextSymbols</code>	<code>IntT</code>	<code>True</code>
<code>FP_IsOnScreen</code>	<code>IntT</code>	<code>True</code>

Pages

Frame documents have three kinds of visible pages: body pages, master pages, and reference pages.

What the user sees

With FrameMaker, the user can change any of the visible pages.

Body pages

Body pages are what a user normally thinks of as the document's pages. They organize the text and graphic objects that appear in the body of a document.

Master pages

Master pages control the layout of body pages. Each body page is associated with one master page, and each master page can be associated with zero or more body pages. A master page provides the following for the body page:

- The text frame layout, which defines the number, size, and placement of the page's text frames and the column layout within each text frame
- The page background, which includes graphic objects and text frames (such as headers and footers) with unnamed flows

By default, single-sided documents have at least one master page, which is named Right. Double-sided documents have two master pages, named Right and Left. FrameMaker products allow you to add your own custom master pages to both single-sided and double-sided documents.

When a FrameMaker product adds a body page, it uses a left or right master page object as a background. It also copies all the text frames with named flows from that master page to the body page. Once the FrameMaker product copies these text frames to the body page, they are independent of the text frames on the master page from which they were copied. If you change the text frames, it does not affect the master page.

A body page's background appears when you view a body page on the screen or print it. However, the background is part of the master page and not the body page. The FrameMaker product superimposes the body page on the background for displaying and printing. If you go to the master page and change the graphic objects that constitute the background, the changes appear when you view or print the body pages associated with the master page.

Reference pages

Reference pages can contain *named graphic frames*. Named graphic frames provide decoration, such as a thick line ruling, for paragraphs in the body pages. The Paragraph Designer provides two settings, Frame Above and Frame Below, that allow you to specify the named graphic frames you want to appear above or below a paragraph.

Reference pages can also contain special flows that control the appearance of generated documents. For example, a Table of Contents document normally has a flow named TOC on one of its reference pages.

How the API represents pages

FrameMaker represents body pages, master pages, and reference pages with `FO_BodyPage`, `FO_MasterPage`, and `FO_RefPage` objects, respectively.

In addition to these pages, a document can also have a *hidden* page, which stores hidden conditional text. The user cannot see or directly modify hidden pages. FrameMaker represents each hidden page with an `FO_HiddenPage` object.

Page objects have a number of common properties. These properties specify the following characteristics of a page:

- The dimensions of the page
- Its type (body, master, reference, or hidden)
- Its numbering
- IDs of the objects that represent its page frame and its sibling pages

A page object does not actually contain the text and graphic objects that appear on a page. Instead, it has a property named `FP_PageFrame`, which specifies the ID of a *page frame*. A page frame is an invisible unanchored frame whose dimensions match those of the page. (For more information on unanchored frames, see “Graphic objects” on page 92.) The API represents a page frame with an `FO_UnanchoredFrame` object. This `FO_UnanchoredFrame` object has properties that specify the IDs of the first and last objects in the linked list of API graphic objects that appear directly on the page.

Suppose you create a body page with a single text frame as shown in Figure 2-2.

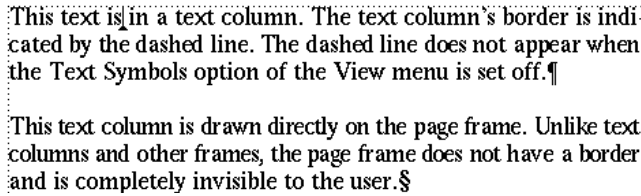


Figure 2-2 *Single text frame on a body page*

The API represents the body page in Figure 2-2 with the objects shown in Figure 2-3. The `FO_BodyPage` object does not have a property that specifies the ID of the `FO_TextFrame` object. Instead, it has a property, named `FP_PageFrame`, that specifies the ID of its page frame (an `FO_UnanchoredFrame` object). The page frame properties, `FP_FirstGraphicInFrame` and `FP_LastGraphicInFrame`, both specify the ID of the `FO_TextFrame` object, since it is the only object that appears directly on the page.

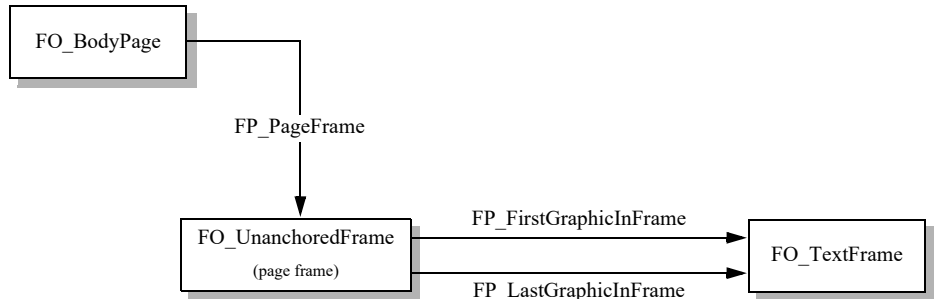


Figure 2-3 *Objects for a body page with a single text frame*

A FrameMaker product automatically creates and destroys the page frame when it creates and destroys the page. The page frame is not accessible to the user. However, as a developer, you need to use it to get the objects on a page.

How the API organizes pages

The API maintains the different types of visible pages in a document (body pages, master pages, and reference pages) in separate linked lists. `FO_Doc` objects have the following properties that specify the first and last page object in each list:

- `FP_FirstBodyPageInDoc` and `FP_LastBodyPageInDoc`
- `FP_FirstMasterPageInDoc` and `FP_LastMasterPageInDoc`
- `FP_FirstRefPageInDoc` and `FP_LastRefPageInDoc`

Each page object has two properties, `FP_PagePrev` and `FP_PageNext`, that specify the IDs of the page objects before and after it in the list. When you delete a page, the API removes the object that represents it and updates the `FP_PagePrev` and `FP_PageNext` properties for all the `FO_Page` objects before and after it.

`FO_Doc` objects also have a property named `FP_CurrentPage` that specifies the ID of the *current page*. The current page is the page that appears on the screen. If more than one page appears on the screen, it is the page that appears with a dark border around it.

Suppose you create a double-sided document that has three body pages, two master pages (Left and Right), and a single reference page, as shown in Figure 2-4. The current page is the Right master page.

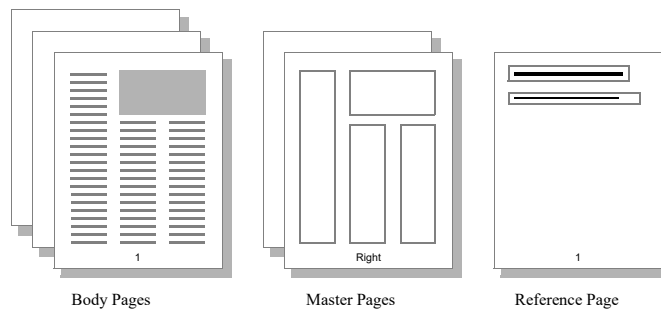


Figure 2-4 Document with body, master, and reference pages

FrameMaker products organize the objects as shown in Figure 2-5.

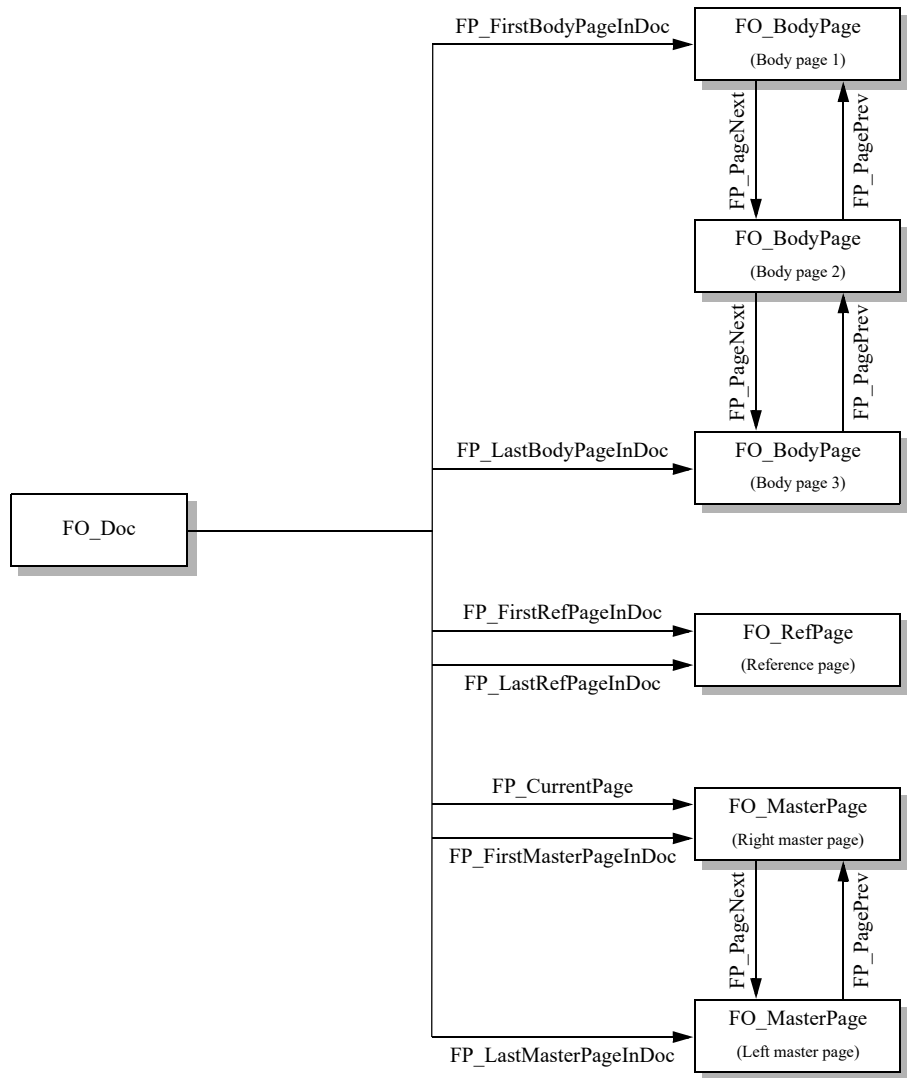


Figure 2-5 Example document and page objects

How the API represents hidden pages

If the user chooses to hide conditional text in the document, the FrameMaker product adds a hidden page to the document to store the hidden text. A document can have only one hidden page. FrameMaker products represent the hidden page with an `FO_HiddenPage` object. The `FO_Doc` property `FP_HiddenPage` specifies its ID. The hidden page has only one text flow, named `HIDDEN`. For more information on how the API represents hidden conditional text, see page 122.

How the API represents master pages

Both single-sided and double-sided documents have default master pages, named `Right` and `Left`. In single-sided documents, the `Left` master page is not visible to the user. However, you can get and set its properties with the API.

Graphic objects

A graphic object is anything the user can draw with the Tools palette, or an imported graphic.

What the user sees

A graphic object can be:

- An anchored frame, which is a container for graphic objects that is tied to a specific location in text.
- An unanchored frame, which is a container for graphic objects that is not tied to a specific location in text.
- A simple geometric shape, which is a line, an arc, a rectangle, a rounded rectangle, an ellipse, a polyline, or a polygon.
- A group, which is an invisible graphic object that holds together a set of other graphic objects.
- A text line, which is a single line of text that isn't in a paragraph or flow (for more information on text lines, see "Text" on page 114).
- A text frame, which is a container for text in a flow (for more information on text frames, see "Text" on page 114).
- An imported graphic such as a bitmap or a PostScript file created with another application.
- An inset or imported graphic.
- A math equation, which describes a formatted equation.

You can draw a graphic object directly on a page in a document. A graphic object drawn directly on a page doesn't move as you edit the text around it. You can also draw a graphic object inside an anchored or unanchored frame. When you move a frame, all the graphic objects inside it move with it. You can nest frames; that is, you can draw a frame within a frame within a frame.

Draw order

The graphic objects in a frame have a back-to-front order or *draw order* that specifies the order in which the FrameMaker product draws them. By default, the draw order is the same as the order in which you created the objects. When graphic objects overlap, the ones in front (at the end of the draw order) obscure those in back. You can change the draw order by selecting a graphic object and choosing Front or Back from the Tools palette.

Groups

You can create a group from one or more graphic objects. This allows you to manipulate them as a single object. When you resize the group, the FrameMaker product automatically resizes the group's component objects proportionally.

Anchored frames

You can draw graphic objects in anchored frames, which are tied to text symbols named *anchor symbols* (^). An anchor symbol (and its anchored frame) moves with the text to which it is attached. You can specify a variety of parameters that determine where a frame appears in relation to its anchor symbol. For example, it can be below the line containing the anchor symbol or at the bottom of the text frame containing the anchor symbol. Unlike other graphic objects, the anchored frame cannot be drawn directly on a page or into another frame; it can only be created in text.

How the API represents graphic objects

The API represents each type of graphic object with a different type of API object. For example, it represents polygons with `FO_Polygon` objects and text frames with `FO_TextFrame` objects.

All types of *API graphic objects*¹ have properties that provide the following information:

- The graphic object's format—that is, characteristics such as its fill pattern and border width
- The graphic object's location and angle

.....

1. This manual uses the term *API graphic object* to refer to objects (such as `FO_Polygon` and `FO_TextFrame` objects) that the API uses to represent the graphic objects (such as polygons and text frames) that appear on a page.

- IDs of the graphic object's parent, sibling, and child objects

Some format properties do not affect some graphic objects. For example, an `FO_Rectangle` object, like all other objects, has an `FP_ArrowType` property. This property can have a value, but that value will not affect the appearance of the rectangle that the object represents.

All types of API graphic objects also have several properties that are specific to them. For example, `FO_Arc` objects have a property named `FP_Theta` that specifies an arc's theta value.

Suppose you create the arrow shown in Figure 2-6.

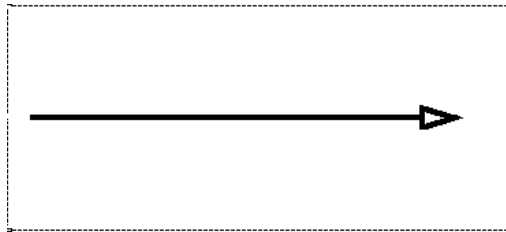


Figure 2-6 Arrow in an unanchored frame

The API represents the arrow with an `FO_Line` object, whose properties include those shown in the following table.

Property	Type	Value
<code>FP_FrameParent</code>	<code>F_ObjHandleT</code>	ID of the frame containing the arrow
<code>FP_Pen</code>	<code>IntT</code>	<code>FV_FILL_BLACK</code>
<code>FP_LocX</code>	<code>MetricT</code>	<code>.25*in</code>
<code>FP_LocY</code>	<code>MetricT</code>	<code>1.125*in</code>
<code>FP_HeadArrow</code>	<code>IntT</code>	<code>True</code>
<code>FP_ArrowType</code>	<code>IntT</code>	<code>FV_ARROW_HOLLOW</code>

How the API organizes graphic objects

The API maintains each API graphic object in at least two linked lists:

- The list of all the API graphic objects in the document

For convenience, the API maintains a linked list of all the API graphic objects in a document. The `FO_Doc` property `FP_FirstGraphicInDoc` specifies the ID of the first object in the list. API graphic objects have a property named `FP_NextGraphicInDoc`, which specifies the ID of the next API graphic object in

the list. If you traverse this list, you will cover every graphic object in a document. The order of the list is completely random.

- The list of API graphic objects in the graphic object's parent frame
Each API graphic object (except an anchored frame and a page frame) has exactly one *parent frame* object. The parent frame is the frame that contains the graphic object. The API maintains a linked list of the child objects in each frame. FO_UnanchoredFrame and FO_AFrame objects have FP_FirstGraphicInFrame and FP_LastGraphicInFrame properties, which specify the first and last objects in the list of their child objects. All API graphic objects have FP_PrevGraphicInFrame and FP_NextGraphicInFrame properties, which specify the objects before and after them in the list. The order of the objects in the linked list is the same as the draw order of the graphic objects in a frame.

Like the frames they represent, API frame objects can be nested: that is, an FO_UnanchoredFrame or FO_AFrame object can be the parent of another FO_UnanchoredFrame object. Every API graphic object (except an object that represents a page frame or an anchored frame) is a descendant of exactly one API page frame object.

Suppose you create a page that contains:

- An unanchored frame that contains an oval, a rectangle, and a text frame with some text in it
- A text line that overlaps the unanchored frame, but is drawn directly on the page

FrameMaker products organize the objects as shown in Figure 2-7.

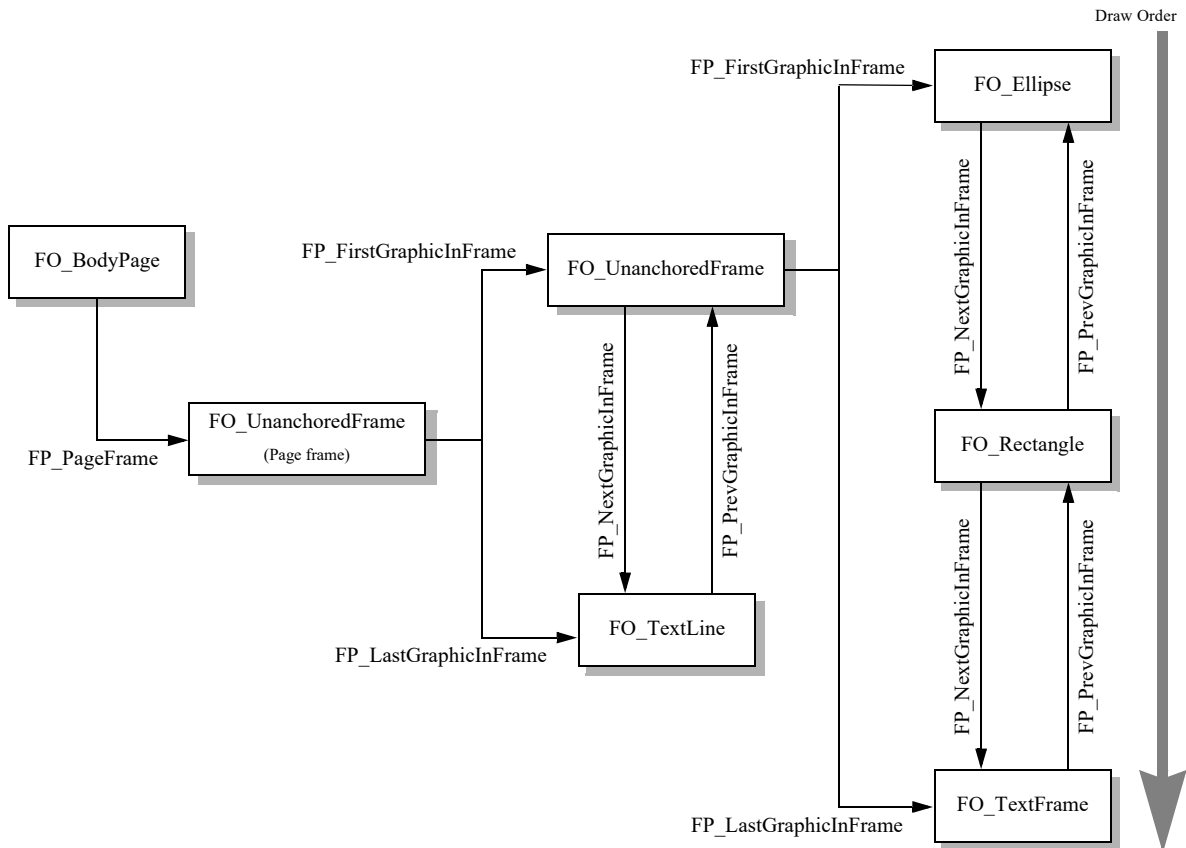


Figure 2-7 API objects that represent a page and the graphic objects on it

How the API represents groups

The API represents a grouped set of graphic objects with an `FO_Group` object. It maintains the objects that constitute a group in a linked list. The `FO_Group` properties, `FP_FirstGraphicInGroup` and `FP_LastGraphicInGroup`, specify the first and last objects in the list. Each graphic object has `FP_PrevGraphicInGroup` and `FP_NextGraphicInGroup` properties, which specify the objects before and after it in the list.

Grouping graphic objects does not affect their position in the linked list of API graphic objects in a frame. That is, it does not affect their position in the draw order. Group objects themselves have an arbitrary position in the draw order.

How the API represents selections of graphic objects

The `FO_Doc` property, `FP_FirstSelectedGraphicInDoc`, specifies the ID of the object that represents the first selected graphic object in a document. If more than one graphic object is selected, the API forms a linked list of the API graphic objects that represent the selected graphic objects. API graphic objects have an `FP_NextSelectedGraphicInDoc` property that specifies the ID of the next selected graphic object. The order of the list is not necessarily the same as the order in which the graphic objects were selected.

Although `FP_FirstSelectedGraphicInDoc` is a document property, you can only select graphic objects that are within the same frame.

Flows

FrameMaker products use flows to connect text frames in a document.

What the user sees

A flow tells the FrameMaker product where to put additional text when a text frame is full. In a simple document, there may be only one flow associated with the body pages. In complex documents such as newsletters, you may create multiple flows that have connected text frames on different pages.

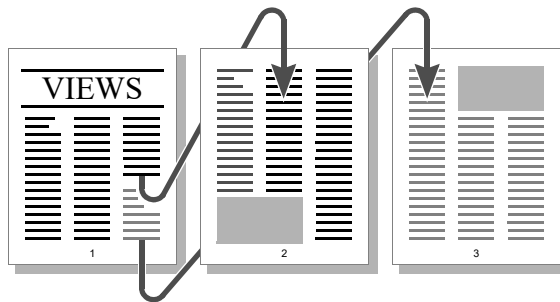


Figure 2-8 *Multiflow document*

Flows have an `Autoconnect` setting that controls whether the FrameMaker product generates a new page when you fill the last text frame of a flow. If `Autoconnect` is on, the FrameMaker product creates a new page and connects a text frame on the new page with the text frame on the previous page.

Main flows

Each document has a main flow. Normally, the FrameMaker product treats the main flow like any named flow in the document. However, there are cases when the FrameMaker product treats the main flow specially:

- When you generate a table of contents or an index, the FrameMaker product puts the generated text into the main flow of the generated document.
- When you run Compare Documents, the FrameMaker product puts the Summary text into the main flow.

Usually the main flow is the default flow for the current language. For example, if the current language is English, the main flow is A.

If there are several Autoconnect flows in the document with the default flow tag, the main flow is the one in the backmost text frame on the frontmost body page.

How the API represents flows

The API represents a flow with an `FO_Flow` object, whose properties provide the following information:

- The flow's format characteristics, such as the feathering and whether Autoconnect is enabled
- The IDs of the first and last `FO_TextFrame` objects in the flow
- The ID of the next `FO_Flow` object in the document

How the API organizes flows

The API maintains a document's `FO_Flow` objects in a linked list.

The `FO_Doc` property, `FP_FirstFlowInDoc`, specifies the ID of the first `FO_Flow` object in the list. `FO_Flow` objects have a property named `FP_NextFlowInDoc`, which specifies the next `FO_Flow` object in the list. The order of the list is random; it does *not* correspond to the order in which the flows appear in the document.

The API also maintains the objects that represent a flow's text frames in a linked list. The `FO_Flow` properties, `FP_FirstTextFrameInFlow` and `FP_LastTextFrameInFlow`, specify the first and last `FO_TextFrame` objects in the list. Each `FO_TextFrame` object has an `FP_PrevTextFrameInFlow` property and a `FP_NextTextFrameInFlow` property, which specify the previous and next `FO_TextFrame` objects in the list. For more information on how flows, text frames, and paragraphs are organized, see "How the API organizes paragraphs" on page 106.

Suppose you create the document shown in Figure 2-9. The document has two flows: a main flow, named A, and a second flow that is unnamed. The A flow connects a two-column text frame on the first page and a two-column text frame on the second page. The unnamed flow appears only on the first page and has only one text frame.

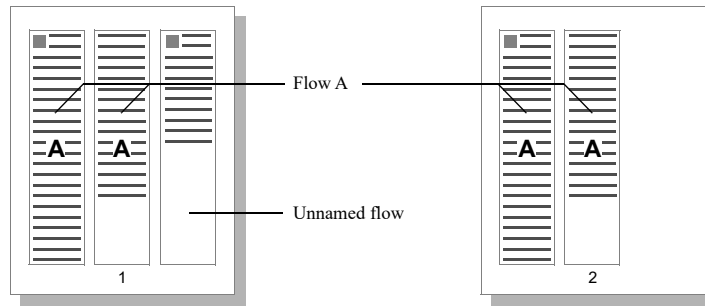


Figure 2-9 Document with a named and an unnamed flow

Figure 2-10 shows how the API organizes the objects that represent the flows and text frames shown in Figure 2-9.

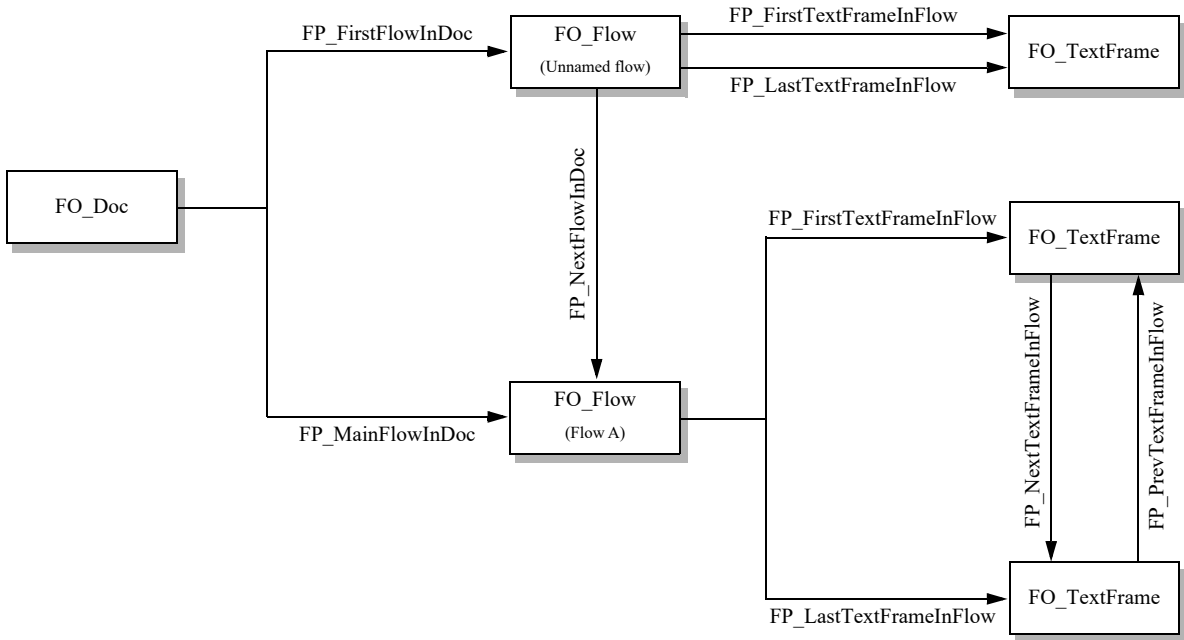


Figure 2-10 Objects that represent a document with two flows

Although the `FP_FirstFlowInDoc` property specifies the `FO_Flow` object for the unnamed flow in Figure 2-10, it could also specify the `FO_Flow` object that represents Flow A. The determination of which flow is first in the list is completely random.

The API uses `FO_SubCol` objects to represent the column formatting of text as follows:

- Contiguous paragraphs in each column of a text frame are within a single `FO_SubCol` object.
- Contiguous paragraphs within sidehead area are within a single `FO_SubCol`.
- Each contiguous series of paragraphs that spans columns and/or sideheads is represented by a single `FO_SubCol` object.
- Following paragraphs that do not span columns and sideheads begin a new group of `FO_SubCol` objects. For example, Figure 14.11 shows a page that has seven `FO_SubCol` objects: two groups of three, plus one for the heading that spans all columns..

The `FO_TextFrame` properties, `FP_FirstSubCol` and `FP_LastSubCol`, specify the first and last `FO_SubCol` objects in a specific text frame. Each `FO_SubCol` object has an `FP_PrevSubCol` property and a `FP_NextSubCol` property, which specify the previous and next `FO_SubCol` objects in the flow. Each `FO_SubCol` object also has a `FP_ParentTextFrame` property, which specifies the text frame it is in. If a text frame has only one column, its `FP_FirstSubCol` and `FP_LastSubCol` properties both specify the ID of the `FO_SubCol` object that represents it.

Figure 2-11 shows how the API organizes the objects that represent the two-column text frame on the first page of the document in Figure 2-9.

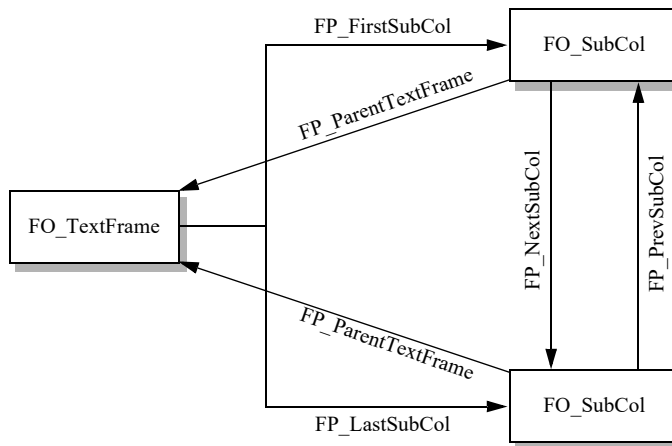


Figure 2-11 Objects that represent a text frame with two columns

In structured FrameMaker, flows can be structured. If a flow is structured, the `FO_Flow` object that represents it has a `FP_HighestLevelElement` property that specifies the ID of the root element.

For information on how the API organizes paragraphs and text in flows, text frames, and columns, see “How the API organizes paragraphs” on page 106.

Paragraph Catalog formats

Each document has a Paragraph Catalog containing one or more Paragraph Catalog formats.

What the user sees

Each Paragraph Catalog format specifies aspects of a paragraph's appearance, such as its indents, line spacing, and default font. Each format has a name or tag, which usually corresponds to a type of paragraph, such as title, body, or heading. To make a paragraph's appearance conform to a format, you apply the format to the paragraph. You can apply the same format to multiple paragraphs to ensure consistency in your document.

You can add formats to the Paragraph Catalog or modify or delete formats that are already in it.

How the API represents Paragraph Catalog formats

FrameMaker represents each Paragraph Catalog format with an `FO_PgfFmt` object, whose properties provide the following information:

- The name of the paragraph format
- Formatting information
- The ID of the next `FO_PgfFmt` object in the document

Suppose you create the paragraph format described in the Paragraph Designer in Figure 2-12.

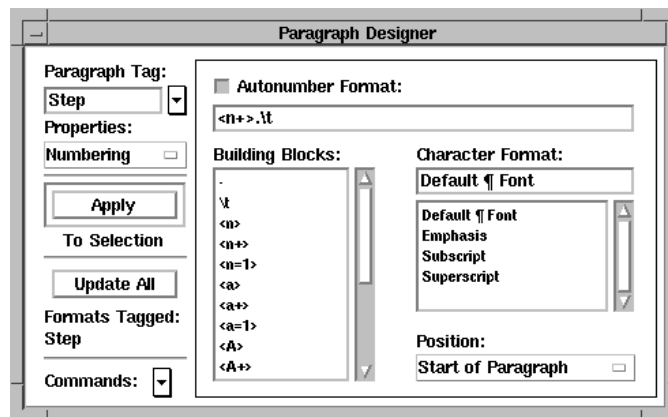


Figure 2-12 *Paragraph Designer*

The API represents this paragraph format with an `FO_PgfFmt` object. The following table lists some of its properties.

Property	Type	Value
<code>FP_Name</code>	<code>StringT</code>	<code>Step</code>
<code>FP_AutoNumString</code>	<code>StringT</code>	<code><n+>.\t</code>
<code>FP_AutoNumChar</code>	<code>StringT</code>	<code>" "</code>
<code>FP_PgfIsAutoNum</code>	<code>IntT</code>	<code>True</code>
<code>FP_NextPgfFmtInDoc</code>	<code>F_ObjHandleT</code>	ID of the next Paragraph Catalog format in the document's list of Paragraph Catalog formats

The `FP_AutoNumChar` property specifies the character format to be applied to the string specified by `FP_AutoNumString`. When the default font is used, `FP_AutoNumChar` is set to a null string (`" "`).

How the API organizes Paragraph Catalog formats

The API organizes the `FO_PgfFmt` objects in a document in a linked list. The `FO_Doc` property, `FP_FirstPgfFmtInDoc`, specifies the first `FO_PgfFmt` object in the list. `FO_PgfFmt` objects have an `FP_NextPgfFmtInDoc` property, which specifies the ID of the next `FO_PgfFmt` object in the list. The order of the list does *not* correspond with the order in which the formats appear in the Paragraph Catalog.

Paragraphs

A paragraph can be a body of text, a title, or an item in a list.

What the user sees

You can type a paragraph in a text frame, a footnote, or a table cell.

Every paragraph has a paragraph format consisting of:

- A tag, which is the name of a format stored in the Paragraph Catalog
- Formatting information, which is the same information that a Paragraph Catalog format provides, such as indents and leading

Every paragraph starts with a tag and formatting information that matches a Paragraph Catalog format. There are several ways you can change a paragraph's format:

- Apply a different Paragraph Catalog format to the paragraph.
When you do this, the FrameMaker product changes the paragraph's formatting information to match that of the Paragraph Catalog format. This process is known as tagging.
- Change the paragraph's formatting information.
This does not affect the Paragraph Catalog format that you tagged the paragraph with. For example, if you tag a paragraph with a tag named `indentbody` that specifies a 1-inch indent and subsequently change the paragraph's indent to 2 inches, the `indentbody` format and other paragraphs tagged as `indentbody` still have a 1-inch indent. This change is a format override, and it applies only to that paragraph instance.
- Change the Paragraph Catalog format's formatting information.
FrameMaker products allow you to update all the paragraphs that are tagged with the format you changed. You can choose whether you want to retain format overrides when FrameMaker updates all paragraphs in the document with the same tag.

How the API represents paragraphs

FrameMaker products represent each paragraph with an `FO_Pgfl` object, whose properties provide the following information:

- The ID of the text frame and text column containing the paragraph
- The paragraph's formatting information (the same set of properties that a Paragraph Catalog format provides)
- The paragraph's tag
- The IDs of sibling `FO_Pgfl` objects
- A flag indicating whether the paragraph has been successfully spell-checked since the last change was made to it

Each paragraph object also contains an `F_TextItemsT` structure, which represents the text in the paragraph. For more information about text and the `F_TextItemsT` structure, see "How the API represents text" on page 114.

Suppose you create the paragraph shown in Figure 2-13.

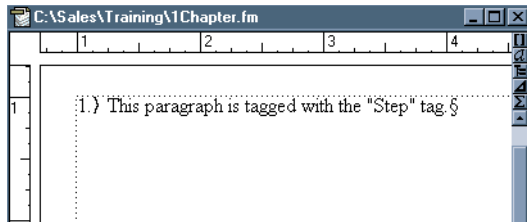


Figure 2-13 A paragraph

The API represents the paragraph with an `FO_Pgf` object. The following table lists some of its properties.

Property	Type	Value
<code>FP_Name</code>	<code>StringT</code>	Step
<code>FP_AutoNumString</code>	<code>StringT</code>	<n+>.\t
<code>FP_PgfIsAutoNum</code>	<code>IntT</code>	True
<code>FP_PgfNumber</code>	<code>StringT</code>	1.
<code>FA_LeftIndent</code>	<code>MetricT</code>	0
<code>FP_InTextFrame</code>	<code>F_ObjHandleT</code>	ID of the text frame the paragraph starts in
<code>FP_InTextObj</code>	<code>F_ObjHandleT</code>	ID of the subcolumn (<code>FO_SubCol</code> object) the paragraph starts in

A paragraph's `FP_InTextObj` property does not always specify the ID of a subcolumn. If the paragraph appears in a table cell, it specifies the ID of the `FO_Cell` object representing the cell. If the paragraph appears in a footnote, `FP_InTextObj` specifies the ID of the `FO_Fn` object representing the footnote.

How to apply formats to paragraphs

To apply a format from the paragraph format catalog to a specific paragraph object, first get the ID of the `FO_Pgf` object in question. Then loop through the document looking for the `FO_PgfFmt` object with a name that matches the tag you want to apply to the paragraph. Then use `F_ApiGetProps()` to get the list of properties from the `FO_PgfFmt` object, and use `F_ApiSetProps()` to set the property list to the `FO_Pgf` in question.

How the API organizes paragraphs

The API maintains `FO_Pgf` objects in two linked lists:

- The list of all `FO_Pgf` objects in a document
- The list of `FO_Pgf` objects in a flow

The list of paragraphs in a document

The `FO_Doc` property, `FP_FirstPgfInDoc`, specifies the first `FO_Pgf` object in the list of `FO_Pgf` objects in a document. Each `FO_Pgf` object has an `FP_NextPgfInDoc` property, which specifies the next `FO_Pgf` object in the list. The order of the list of `FO_Pgf` objects in a document does *not* necessarily correspond to the actual order of the paragraphs in the document.

The list of paragraphs in a flow

`FO_Flow` objects do not have a property that specifies the first `FO_Pgf` object in a flow. To find the first `FO_Pgf` object in the flow, you must find the first `FO_TextFrame` object in the flow. Then you must get the `FO_Pgf` object specified by the `FO_TextFrame` object's `FP_FirstPgf` property. In some cases, the first text frame in the flow may not contain any paragraphs. You must traverse subsequent text frames and check them to see if they contain any paragraphs.

Each `FO_Pgf` object has `FP_PrevPgfInFlow` and `FP_NextPgfInFlow` properties, which specify the IDs of the `FO_Pgf` objects before and after it in the flow. To get the paragraphs in a flow in order, you traverse these properties.

It is possible for a paragraph to begin in one text frame and end in another. When this happens, the ID of the `FO_Pgf` is specified by the `FP_LastPgf` property of the text frame in which it begins *and* the `FP_FirstPgf` property of the text frame in which it ends.

Suppose you create two text frames and connect them with a flow. The first text frame has two paragraphs in it; the second paragraph continues into the next text frame as shown in Figure 2-14.

This is paragraph 1, which begins and ends in text column 1.¶	but ends in text column 2 ¶
This is paragraph 2 which begins in text column 1	This is paragraph 3, which begins and ends in text column 2.§

Figure 2-14 *Flow with two text frames*

The API organizes the objects that represent the flow, text frames, and paragraphs as shown in Figure 2-15.

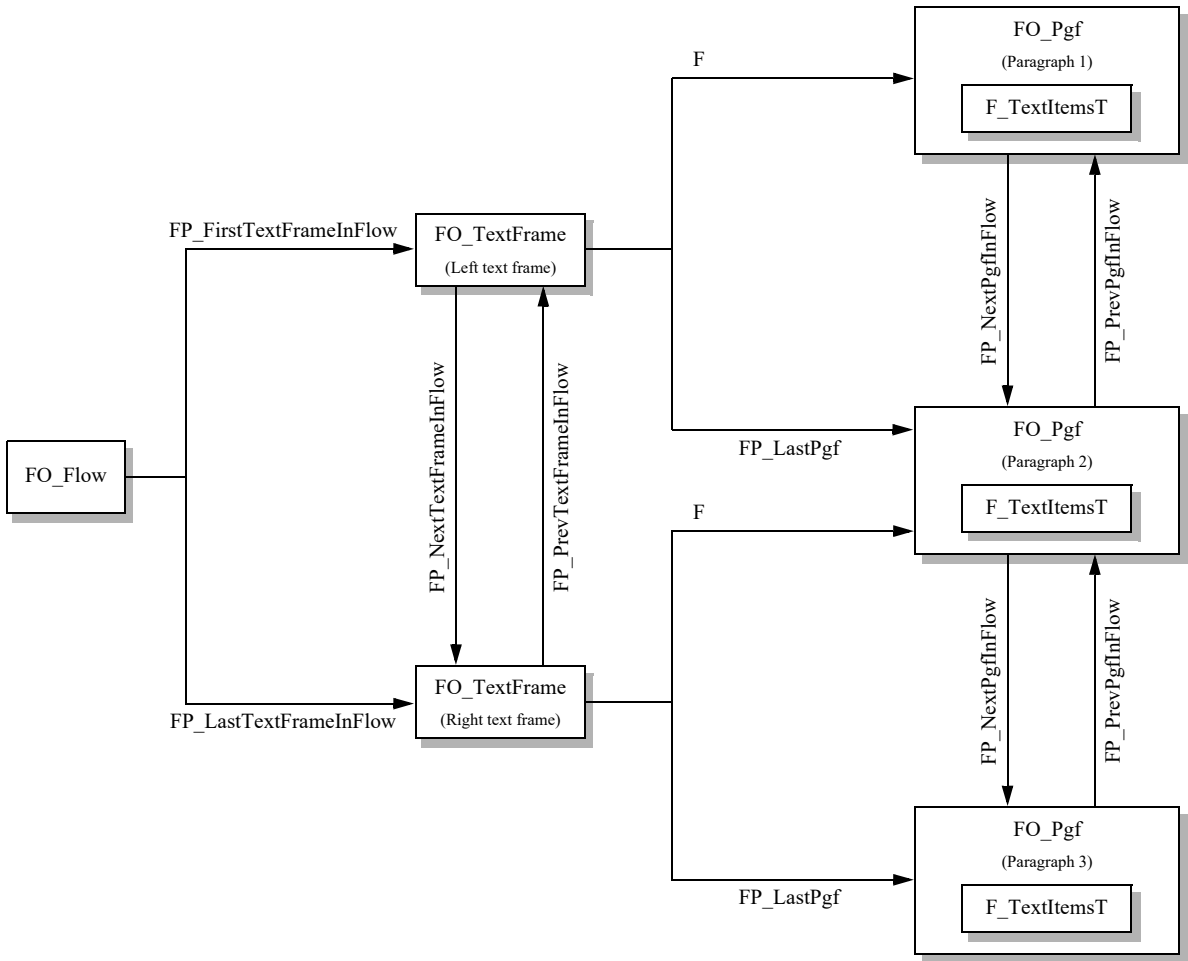


Figure 2-15 Objects that represent a flow with text frames and paragraphs

Like text frames, subcolumns (FO_SubCol objects) have FP_FirstPgf and FP_LastPgf properties, which specify the first and last paragraphs in them. If a paragraph begins in one text column and ends in another, the paragraph's ID is specified by the FP_LastPgf property of the subcolumn in which it begins *and* the FP_FirstPgf property of the subcolumn in which it ends.

Character Catalog formats

Each document has a Character Catalog containing one or more character formats.

What the user sees

Each character format has a name (or tag), which usually corresponds to a type of text, such as Emphasis, Usertype, or Booktitle. It also contains information about how text should look, such as its font family, weight, and angle. To make the appearance of a set of characters conform to a Character Catalog format, you apply the format to the set of characters.

When you apply a character format to a set of characters, it overrides the default font setting of the paragraph format. If you reapply a Paragraph Catalog format to the paragraph, it does not affect the format of the characters that you previously tagged with the character format.

FrameMaker products allow you to create character formats that override only some properties of the text to which they are applied. To leave one of the current text properties intact, you set the corresponding character format property to As Is. The Character Designer indicates the As Is state with the words As Is or a grayed (or stippled) checkbox.

You can add character formats to the Character Catalog or modify or delete formats that are already in it.

How the API represents Character Catalog formats

FrameMaker products represent each Character Catalog format with an `FO_CharFmt` object, whose properties provide the following information:

- The name of the character format
- Character formatting information, such as the font family, angle, and weight
- Whether the character format's formatting overrides the default formatting of the text that the format is applied to
- The ID of the next `FO_CharFmt` object in the document

How the API represents fonts

FO_Session objects have properties (such as FP_FontFamilyNames) that provide arrays of the names of the font families, variations, angles, and weights available in the current session. These lists are referenced by F_StringsT structures. F_StringsT is defined as:

```
typedef struct {
    UIntT len; /* Number of strings */
    StringT *val; /* Array of strings */
} F_StringsT;
```

For example, if Bold and Regular are the only font weights available in the current session, the fields of the F_StringsT structure specified by the FO_Session property, FP_FontWeightNames, have the following values:

```
len: 3
val: { "<None>", "Regular", "Bold" }
```

To set a character format's weight to Bold in this session, you set its FP_FontWeight property to 2.

For more information on session font properties, see “How the API indicates which fonts are available in a session” on page 72.

You can also use the following properties to specify a font:

- FP_FontPlatformName specifies a font name that uniquely identifies the font on the Windows platform.
- FP_FontPostScriptName specifies the name given to a font when it is sent to a PostScript printer (specifically, the name that is passed to the PostScript FindFont operator before any font coordination operations).

The PostScript name is unique for all PostScript fonts, but may not be available for fonts that have no PostScript version.

FrameMaker products ignore the following keywords in PostScript names:

```
83pv
90pv
90ms
Ext
Add
NWP
```

The FP_FontPlatformName property specifies a platform-specific ASCII string that uniquely identifies a font for a particular platform. The string consists of several fields separated by periods.

On Windows, the string you specify for `FP_FontPlatformName` has the following syntax:

```
W.FaceName.ItalicFlag.Weight.Variation
```

This field	Represents
<code>W</code>	Platform designator
<code>FaceName</code>	Windows face name (for more information, see your Windows documentation)
<code>ItalicFlag</code>	Whether font is italic; you can use one of the following flags: I (Italic) R (Regular)
<code>Weight</code>	Weight classification, for example 400 (Regular) or 700 (Bold)

The following strings are valid representations of the Windows font, Helvetica Narrow Bold Oblique:

```
W.Helvetica-Narrow.I.700
```

```
W.Helvetica.I.700.Narrow
```

When reading in a document, a FrameMaker product determines a font name by checking font properties in the following order:

- `FP_FontPlatformName`
- Combination of `FP_FontFamily`, `FP_FontVariation`, `FP_FontWeight`, and `FP_FontAngle`
- `FP_FontPostScriptName`

Your clients do not need to use all three methods to change fonts. You should always specify the PostScript name, if it is available.

How the API represents As Is settings

`FO_CharFmt` objects use two properties to represent a font characteristic: one to represent the characteristic's As Is state and one to represent the characteristic itself.

For example, `FP_UseFontWeight` specifies whether the character format's font weight overrides the default font weight of the text that the format is applied to. `FP_FontWeight` specifies the character format's font weight. If `FP_UseFontWeight` is `True`, the font weight specified by `FP_FontWeight` overrides the default font weight for the text. If `FP_UseFontWeight` is `False` (As Is), `FP_FontWeight` does not affect the text's font weight.

If an `FP_UseCharacteristic` property is `False`, the character format’s property list includes only the `FP_UseCharacteristic` property. It doesn’t include the `FP_Characteristic` property for the characteristic (since this property is not used).

Suppose you create the character format specified in the Character Designer in Figure 2-16.

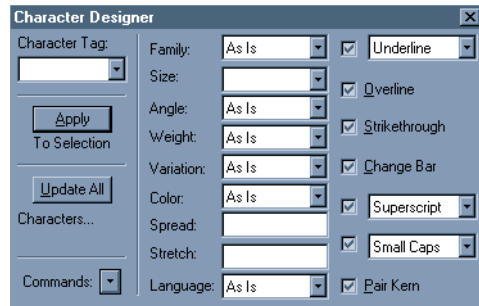


Figure 2-16 Character Designer

The API represents the character format with an `FO_CharFmt` object. The following table lists some of its properties.

Property	Type	Value
<code>FP_CharTag</code>	<code>StringT</code>	<code>booktitle</code>
<code>FP_FontAngle</code>	<code>IntT</code>	Index of Italic font angle
<code>FP_UseFontAngle</code>	<code>IntT</code>	<code>True</code>
<code>FP_UseFontFamily</code>	<code>IntT</code>	<code>False</code>
<code>FP_UseFontVariation</code>	<code>IntT</code>	<code>False</code>
<code>FP_UseFontWeight</code>	<code>IntT</code>	<code>False</code>
<code>FP_UseUnderline</code>	<code>IntT</code>	<code>False</code>
<code>FP_FontSize</code>	<code>MetricT</code>	<code>36*pts</code>
<code>FP_UseFontSize</code>	<code>IntT</code>	<code>True</code>

This character format overrides the default font angle setting and the size of the text to which it is applied. It does not override any of the text’s other default characteristics.

How the API organizes Character Catalog formats

The API organizes the formats in a document's Character Catalog in a linked list. `FO_Doc` objects have an `FP_FirstCharFmtInDoc` property that specifies the first `FO_CharFmt` object in the list. `FO_CharFmt` objects have an `FP_NextCharFmtInDoc` property, which specifies the ID of the next `FO_CharFmt` object in the list. The order of the list does *not* correspond to the order in which the formats appear in the Character Catalog.

Condition Formats

FrameMaker products provide condition formats that allow the user to selectively show or hide text in a document.

What the user sees

To selectively show and hide text, you create a condition format (or tag) and apply it to selections of text. For example, you can create a condition tag named Comment and apply it to all the comments you add to a document. You can then instruct the FrameMaker product to hide all the text with the Comment tag when you print a final draft of the document. A document can have multiple condition tags.

FrameMaker products allow you to specify a format override, or a special style and color for a condition. For example, you can make all text tagged with the Comment condition underlined and red.

How the API represents condition formats

The API represents each condition format with an `FO_CondFmt` object, whose properties provide the following information:

- The condition name
- Whether text tagged with the condition is currently visible
- The format overrides
- The ID of the next condition format in the document

The API represents the condition setting of a location in text as a *text property*. For more information on text properties, see “How the API represents text” on page 114.

If you choose to hide a condition tag, the FrameMaker product moves text with that tag to a hidden page and replaces it with markers. For more information on hidden conditional text, see “How the API represents hidden conditional text” on page 122.

Suppose you create the condition tag specified in the Edit Condition Tag dialog box shown in Figure 2-17.

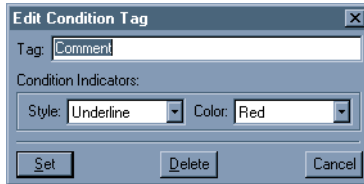


Figure 2-17 Edit Condition Tag dialog box

The API represents the condition with an `FO_CondFmt` object. The following table lists some of its properties.

Property	Type	Value
<code>FP_Name</code>	<code>StringT</code>	Comment
<code>FP_NextCondFmtInDoc</code>	<code>F_ObjHandleT</code>	ID of next condition in list of document’s conditions
<code>FP_CondFmtIsShown</code>	<code>IntT</code>	True
<code>FP_SepOverride</code>	<code>F_ObjHandleT</code>	ID of the <code>FO_Color</code> object that represents red color
<code>FP_StyleOverride</code>	<code>IntT</code>	<code>FV_CN_SINGLE_UNDERLINE</code>
<code>FP_UseSepOverride</code>	<code>IntT</code>	True

How the API organizes condition formats

The API organizes a document’s `FO_CondFmt` objects in a linked list. The `FO_Doc` property `FP_FirstCondFmtInDoc` specifies the first `FO_CondFmt` object in the list. `FO_CondFmt` objects have a property named `FP_NextCondFmtInDoc`, which specifies the ID of the next `FO_CondFmt` object in the list. The order of the list does *not* correspond to the order in which the formats appear in the Conditional Text window.

Text

The user can type text into a text line or a paragraph in a text frame, table cell, or footnote.

What the user sees

FrameMaker products allow you to insert things, such as anchored frames, footnotes, tables, and cross-references into text. The point at which you insert these things is called an anchor. FrameMaker products represent an anchor with an anchor symbol (^) on the screen. This symbol is not visible if the Text Symbols view option is turned off. The anchor moves with the text to which it is attached.

All text has a set of properties that specify the following information about it:

- A tag, or the name of a character format stored in the Character Catalog
- Formatting information (the same information that a Character Catalog format provides, such as the font family and size)
- A set of conditional text formats that apply to it

These properties are called *text properties*.

Just as you can override a Paragraph Catalog tag by changing an individual paragraph's format, you can also override a Character Catalog tag by changing the properties of a selection of text.

You can also apply one or more conditions to a selection of text. This allows you to hide or display the text for particular versions of a document.

How the API represents text

The API represents the text in each paragraph or graphic text line with an `F_TextItemsT` structure, which is defined as:

```
typedef struct {
    UIntT len; /* The number of text items */
    F_TextItemT *val; /* Array of text items */
} F_TextItemsT;
```

The API represents an individual text item with an `F_TextItemT` structure, which is defined as:

```
typedef struct
{
    IntT offset; /* Characters from the beginning */
    IntT dataType; /* The type of text item, e.g. FTI_String */
    union {
        StringT sdata; /* String if type is FTI_String */
        IntT idata; /* An ID if the item specifies an object */
    } u;
} F_TextItemT;
```

The `offset` value specifies the distance between the start of the text item and the beginning of the text line or paragraph. This distance is measured in the number of characters (both regular characters and anchor symbols).

Each of the following constitutes a separate text item:

- A string of characters with common text properties

A text item can contain a string that is as long as a line of text. However, the API uses a separate text item for each section of the text that has different text properties. If a single property (such as the font weight, font angle, or condition format) is different, the API starts a new text item. So a single line of text may require several text items to represent it.
- The beginning or end of a line, paragraph, flow, column, page, or structural element

The API uses text items to indicate the beginning or end of the various entities that organize text. Most of these text items specify the ID of an object. Text items that indicate the end of a line specify whether the line end is a regular, hyphenated, or hard line end.
- An anchor for a table, footnote, marker, cross-reference, variable, or anchored frame

The API represents tables, footnotes, markers, cross-references, variables, and anchored frames with separate objects. It uses a text item to represent the anchor for each of these entities. The text item specifies the ID of the object. For example, the API represents a table with an `FO_Tbl` object. It uses a table anchor (`FTI_TblAnchor`) text item to indicate where the table occurs in the text.
- A text properties change

This type of text item identifies the point in text at which the text properties change. It specifies flags that indicate which text properties differ from the properties of the text immediately preceding the text item.

The following table lists the values the `F_TextItemT.dataType` field can have and the types of data the corresponding text item provides.

Text item type (dataType)	What the text item represents	Text item data
<code>FTI_TextObjId</code>	The object to which the offsets of all the text items are relative	ID of an <code>FO_Pgf</code> , <code>FO_Cell</code> , <code>FO_TextLine</code> , <code>FO_TiApiClient</code> , <code>FO_TiFlow</code> , <code>FO_TiText</code> , or <code>FO_TiTextTable</code>
<code>FTI_String</code>	A string of characters with the same condition and character format	A character string
<code>FTI_LineBegin</code>	The beginning of a line	Nothing
<code>FTI_LineEnd</code>	The end of a line and the line end type	If the line end is a normal line end, 0; if it is a forced line end, the <code>FTI_HardLineEnd</code> flag is set; if it is a hyphen line end, the <code>FTI_HyphenLineEnd</code> flag is set
<code>FTI_PgfBegin</code>	The beginning of a paragraph	ID of an <code>FO_Pgf</code>
<code>FTI_PgfEnd</code>	The end of a paragraph	ID of an <code>FO_Pgf</code>
<code>FTI_FlowBegin</code>	The beginning of a flow	ID of an <code>FO_Flow</code>
<code>FTI_FlowEnd</code>	The end of a flow	ID of an <code>FO_Flow</code>
<code>FTI_PageBegin</code>	The beginning of a page	ID of an <code>FO_BodyPage</code> , <code>FO_HiddenPage</code> , <code>FO_MasterPage</code> , <code>FO_RefPage</code>
<code>FTI_PageEnd</code>	The end of a page	ID of an <code>FO_BodyPage</code> , <code>FO_HiddenPage</code> , <code>FO_MasterPage</code> , <code>FO_RefPage</code>
<code>FTI_TextFrameBegin</code>	The beginning of a text frame	ID of an <code>FO_TextFrame</code>
<code>FTI_TextFrameEnd</code>	The end of a text frame	ID of an <code>FO_TextFrame</code>

Text item type (dataType)	What the text item represents	Text item data
FTI_SubColBegin	The beginning of a column	ID of an FO_SubCol
FTI_SubColEnd	The end of a column	ID of an FO_SubCol
FTI_FrameAnchor	An anchored frame	ID of an FO_AFrame
FTI_FnAnchor	A footnote	ID of an FO_Fn
FTI_TblAnchor	A table	ID of an FO_Tbl
FTI_MarkerAnchor	A marker	ID of an FO_Marker
FTI_XRefBegin	The beginning of a cross-reference instance	ID of an FO_XRef
FTI_XRefEnd	The end of a cross-reference instance	ID of an FO_XRef
FTI_VarBegin	The beginning of a variable instance	ID of an FO_Var
FTI_VarEnd	The end of a variable instance	ID of an FO_Var
FTI_TextInsetBegin	The beginning of a text inset	ID of an FO_TiApiClient, FO_TiFlow, FO_TiText, or FO_TiTextTable
FTI_TextInsetEnd	The end of a text inset	ID of an FO_TiApiClient, FO_TiFlow, FO_TiText, or FO_TiTextTable
FTI_ElementBegin	The beginning of a container element	ID of an FO_Element
FTI_ElementEnd	The end of a container element	ID of an FO_Element
FTI_ElemPrefixBegin	The beginning of an element's prefix	ID of an FO_Element
FTI_ElemPrefixEnd	The end of an element's prefix	ID of an FO_Element
FTI_ElemSuffixBegin	The beginning of an element's suffix	ID of an FO_Element
FTI_ElemSuffixEnd	The end of an element's suffix	ID of an FO_Element
FTI_CharPropsChange	A change in the text properties	Flags indicating which properties have changed (see the table below)

Text item type (dataType)	What the text item represents	Text item data
FTI_RubiCompositeBegin	The beginning of a rubi composite (and the beginning of oyamoji text).	ID of an FO_Rubi
FTI_RubiCompositeEnd	The end of a rubi composite.	ID of an FO_Rubi
FTI_RubiTextBegin	The beginning of rubi text (and the end of oyamoji text).	ID of an FO_Rubi
FTI_RubiTextEnd	The end of rubi text.	ID of an FO_Rubi

The following table lists the bit flags that a client can bitwise AND with the `idata` field of an `FTI_CharPropsChange` text item and the types of text property changes each flag indicates. For example, to determine if the font family changed, bitwise AND the `FTF_FAMILY` flag with the `idata` field.

Flag	Meaning
FTF_FAMILY	The font family has changed.
FTF_VARIATION	The font variation has changed.
FTF_WEIGHT	The font weight has changed.
FTF_ANGLE	The font angle has changed.
FTF_UNDERLINING	The underlining has changed.
FTF_STRIKETHROUGH	The strikethrough characteristic has changed.
FTF_OVERLINE	The overline characteristic has changed.
FTF_CHANGEBAR	The change bars have changed.
FTF_OUTLINE	The outline characteristic has changed.
FTF_SHADOW	The shadow characteristic has changed.
FTF_PAIRKERN	The pair kerning has changed.
FTF_SIZE	The font size has changed.
FTF_KERNX	The kern-x characteristic has changed.
FTF_KERNY	The kern-y characteristic has changed.
FTF_SPREAD	The font spread has changed.
FTF_COLOR	The color has changed.
FTF_CHARTAG	The Character Catalog format has changed.

Flag	Meaning
FTF_CAPITALIZATION	The capitalization has changed.
FTF_POSITION	The character position has changed.
FTF_CONDITIONTAG	The condition tag has changed.
FTF_STRETCH	Font stretch value has changed
FTF_LANGUAGE	Character language has changed
FTF_TSUME	Tsume setting has changed
FTF_IIF	An internal flag having to do with asian text. input. If there is a non-zero value for this flag, a front end processor is controlling that text; you should not modify the associated text item.
FTF_ENCODING	The text encoding has changed.
FTF_ALL	OR of all the flags listed above.

Figure 2-18 shows a paragraph and the text items the API uses to represent the paragraph's text.

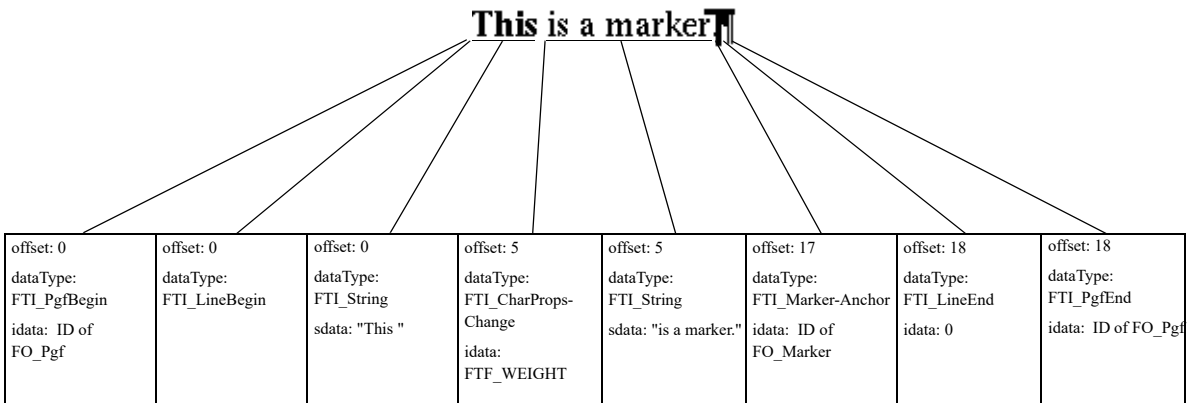


Figure 2-18 Paragraph text and the text items that represent it

There are several important things to note about the text items shown in Figure 2-18:

- Because the string "This " and the string "is a marker ." have different font weights, there are separate text items for them.
- The FTI_CharPropsChange text item indicates that the text properties have changed; the FTF_WEIGHT flag that it specifies indicates that the font weight has changed.

- The marker anchor is counted in the offset.

How the API represents special characters

The API uses the FrameMaker product character set. Some characters are either reserved by the C language or belong to the higher ASCII range. To represent these characters in a string, use octal (\) or hexadecimal (\x) sequences.

Character	Hexadecimal representation	Octal representation
>	\x3e	\76
" (straight double quotation mark)	\x22	\42
“ (left curved quotation mark)	\xd2	\322
” (right curved quotation mark)	\xd3	\323

For a complete list of the characters in the FrameMaker product character set and the corresponding hexadecimal codes, see your Frame product user’s manual. If you are using ANSI C, you can use these hexadecimal codes or their octal equivalents. If you are not using ANSI C, you must use octal (\) sequences.

Suppose you want to represent the following text in the API:

This is an em dash —

If you are not using ANSI C, you must specify the string `This is an em dash \321`. If you are using ANSI C, you can also specify the string `This is an em dash \xd1`.

How the API represents text properties

The `FTI_CharPropsChange` text item only indicates that particular text properties *have* changed. It does not indicate what they have changed to. The API provides a function named `F_ApiGetTextProps()`, which allows you to retrieve the text properties for individual characters in text. You cannot retrieve the text properties for a range or selection of text, because they may be different for individual characters within the selection. You *can*, however, set the text properties for a range of text. For examples of how to get and set text properties, see “Getting and setting text formatting” on page 336.

Suppose you retrieve the text properties at the insertion point shown in Figure 2-19.

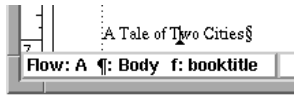


Figure 2-19 *Text containing insertion point*

The following are some of the properties of the text at the insertion point.

Property	Type	Value
FP_CharTag	StringT	booktitle
FP_FontFamily	IntT	Index of Times font (for more information on font name indexes, see “How the API indicates which fonts are available in a session” on page 72)
FP_FontWeight	IntT	Index of Bold font weight
FP_FontAngle	IntT	Index of Regular font angle
FP_InCond	F_IntsT	NULL

If the conditions `Comment` and `MyComment` are applied to the text location, the `FP_InCond` property specifies an `F_IntsT` structure with the following values:

```
len: 2
val: { Comment_ID, MyComment_ID }
```

where `Comment_ID` and `MyComment_ID` are the IDs of the `FO_CondFmt` objects that represent the `Comment` and `MyComment` condition formats.

How the API represents hidden conditional text

The user can choose to hide all the text with a specified condition format. If a document has hidden conditional text, the FrameMaker product automatically adds a hidden page to it. This hidden page is completely invisible to the user. It has a single flow, named `HIDDEN`.

When the user chooses to hide text with a condition format, the FrameMaker product removes each block of text with that condition format and inserts a Conditional Text marker (type 10) in its place. This marker text consists of a plus sign (+) and a five-digit integer. The FrameMaker product places the blocks of hidden text in the `HIDDEN` text flow on the hidden page. The text begins with a Conditional Text marker containing a minus sign (?) and the integer. It ends with another Conditional Text marker containing an equal sign (=) and the integer. If the hidden conditional text doesn't span paragraphs, it appears in one paragraph. If the hidden conditional text spans paragraphs, each paragraph of conditional text constitutes a separate paragraph in the `HIDDEN` flow.

Suppose you have a body page with some conditional text and some unconditional text. The condition tag's format overrides specify that the text appears underlined, as shown in Figure 2-20.

```
This whole paragraph is conditional.¶  
This sentence contains conditional text.¶  
This paragraph is also conditional.¶  
This is a normal paragraph.¶  
§
```

Figure 2-20 Body page with conditional and unconditional text

If you hide the text, the body page appears as shown in Figure 2-21.

```
Marker text: +84974 — This sentence contains text.¶  
                    This is a normal paragraph.¶  
                    §  
                    |  
                    |  
Marker text: +95675   Marker text: +93024
```

Figure 2-21 Body page with the conditional text hidden

If you could see the hidden page and the text in the HIDDEN flow, it would appear as shown in Figure 2-22. The numbers in the markers that represent the hidden conditional text on the body page correspond to the numbers in the markers on the hidden page.

```
Marker text: -84974 — This whole paragraph is conditional.¶ Marker text: =84974  
Marker text: -95675 — Conditional¶ Marker text: =95675  
Marker text: -93024 — This paragraph is also conditional§ Marker text: =93024
```

Figure 2-22 Hidden conditional text on the hidden page

Markers

Markers are anchored objects that store data and associate that data with specific locations in the text. Various features in FrameMaker may refer to a marker, or you can use markers to store data for your FDK clients.

What the user sees

You can use markers to mark entries for a table of contents or an index. A marker's position in text is indicated by a marker symbol. A marker contains text, which appears in the Marker window when you select the marker and choose Marker from the Special menu.

Any number of marker types can be defined for a document; 11 of them are predefined by the FrameMaker product as a standard list of marker types, and the others are defined by the user. The list of defined marker types is saved with the document.

How the API represents markers

The API represents each marker with the following:

- An `FTI_MarkerAnchor` text item that specifies the ID of an `FO_Marker` object
- An `FO_Marker` object

`FO_Marker` properties provide the following information:

- The marker type; the Id of an `FO_MarkerType` object
- The text the marker contains
- The ID of the next `FO_Marker` object in the document
- The location of the marker in text
- The element ID of the marker, if it is a structured marker in a structured document
- If included, the number of a marker type in versions earlier than 5.5; when opening the document in FrameMaker 5.5, this maps the old numbered marker type to the new named `bmarker` type

`FO_MarkerType` properties provide the following information:

- The ID of the next `FO_MarkerType` object in the document
- The name of the marker type, as it appears in the user interface
- The internal name of the marker type (usually the same as the name that appears in the user interface)

- If included, the number of a marker type in versions earlier than 5.5; when opening the document in FrameMaker 5.5, this maps the old numbered marker type to the new named marker type
- Whether the marker type appears in the user interface, whether it is saved with the document, and whether the marker type can be deleted

The `FO_Doc` property, `FP_MarkerTypeNames`, specifies an `F_StringST` structure, which provides the list of marker types available in the current document. The document object also has an `FP_FirstMarkerTypeInDoc` property as an entry into the document's list of marker types.

Given a marker type name, you can use `F_ApiGetNamedObject()` to get the ID of the associated `FO_MarkerType`. The following code returns the ID of the index marker type:

```
...
F_ObjHandleT docId, markerId;

/* Get ID of the active document. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);

markerId = F_ApiGetNamedObject(docId, FO_MarkerType, (StringT)
"Index");
...
```

Figure 2-23 shows an index marker anchor and the text item that represents it.

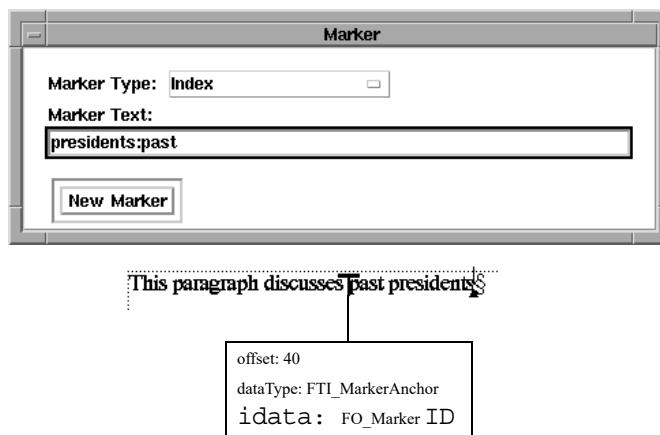


Figure 2-23 A marker anchor and the text item that represents it

The following table lists some of the properties of the `FO_Marker` object specified by `FTI_MarkerAnchor` in Figure 2-23.

Property	Type	Value
<code>FP_MarkerTypeId</code>	<code>F_ObjHandleT</code>	ID of the <code>FO_MarkerType</code> for "Index"
<code>FP_MarkerText</code>	<code>StringT</code>	<code>presidents:past</code>
<code>FP_NextMarkerInDoc</code>	<code>F_ObjHandleT</code>	ID of the next <code>FO_Marker</code> object in the document

Adding marker types to documents

To add a marker type to a document, use `F_ApiNewNamedObject()`. Once you have the new marker type's ID, you can set any properties that you want to be different from the default values.

```
...
F_ObjHandleT docId, myMarkerTypeId;

/* Get ID of the active document. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);

/* Create the marker type. */
myMarkerTypeId = F_ApiNewNamedObject(docId, FO_MarkerType,
                                     (StringT) "MyMarkerType");
...
```

To delete a marker type from a document, pass the document ID and the marker type ID to `F_ApiDelete()`.

Mapping old marker types to named marker types

In versions of FrameMaker earlier than 5.5, marker types were identified by number. Type 1 through Type 10 were reserved for FrameMaker, and given specific names; Header/Footer \$1, Header/Footer \$2, etc. Type 11 through Type 25 were for custom marker types.

`FO_MarkerType` and `FO_Marker` objects have an `FP_OldTypeNum` property that maps the named marker type to what was a numbered marker type in earlier documents. In this way, you can ensure that your client handles markers in legacy data the way you want.

For example, assume your client adds a marker type named `MyMarkerType` to a document, and you set the `FP_OldTypeNum` property of `MyMarkerType` to 11. The user might import or paste text from an older document into the document with `MyMarkerType`. If the older text has markers of type 11 in it, they will come into the new document as `MyMarkerType`.

The standard list of marker types

Every document includes a set of required marker types; Header/Footer \$1, Header/Footer \$2, Index, Comment, Subject, Author, Glossary, Equation, Hypertext Cross-Ref, and Conditional Text. These are required markers, and cannot be deleted.

You can add an existing public marker type to the standard list by setting the name string to the `FP_AddMarkerTypeToStandardMarkers` property of the current session object. Once you add a marker type to this list, it remains for the entire session; you must quit the session to remove it.

```
...
F_ApiSetInt(0, FV_SessionId, FP_OldTypeNum, (IntT) 17);
F_ApiSetString(0, FV_SessionId,
               FP_AddMarkerTypeToStandardMarkers, (StringT)
               "MyMarkerType");
...
```

This example first sets a session integer for `FP_OldTypeNum` to 17. This ensures that for the rest of the current session, markers of type 17 (from earlier documents) will come into new documents as markers of type `MyMarkerType`.

If the the `FP_OldTypeNum` you specify is taken, your new marker type will not be added to the list of standard marker types. To confirm that your marker type was added to the standard list, get the `FP_MarkerNames` property from the `FV_SessionId` object.

It's possible for the `FP_OldTypeNum` you specified to be taken; another API client may have already used that value when assigning a marker type to the standard list. For example, HTML export in FrameMaker 5.5 is performed by a client that adds the HTML Macro marker type to the standard list. The value of that marker's `FP_OldTypeNum` is 11. After that client is initialized, no other clients can use the same value for `FP_OldTypeNum` when assigning a marker to the standard list.

Cross-reference formats

When you insert a cross-reference in a document, you choose a cross-reference format that specifies the information provided by the cross-reference.

What the user sees

A cross-reference format consists of a combination of text and cross-reference building blocks. Cross-reference building blocks are variables that provide information, such as the current page number or filename.

Each document has a catalog of cross-reference formats. You can add or delete formats from this catalog.

How the API represents cross-reference formats

FrameMaker products represent each cross-reference format with an `FO_XRefFmt` object, whose properties provide the following information:

- The name of the cross-reference format
- A string containing the cross-reference's text and cross-reference building blocks
- The ID of the next `FO_XRefFmt` object in the document

Suppose you create a cross-reference format named `See Heading & Page` as shown in Figure 2-24.

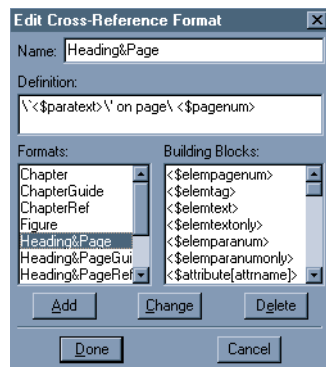


Figure 2-24 A cross-reference format

The following table lists some properties of the `FO_XRefFmt` object that represents this cross-reference format.

Property	Type	Value
<code>FP_Name</code>	<code>StringT</code>	See <i>Heading & Page</i>
<code>FP_Fmt</code>	<code>StringT</code>	See <code>\'<\$paratext>\'</code> on page <code>\ <\$pagenum>.</code>
<code>FP_NextXRefFmtInDoc</code>	<code>F_ObjHandleT</code>	ID of the next <code>FO_XRefFmt</code> object in the document

Cross-references

A cross-reference refers to a specific location, known as a *source*, within the current document or another document. The source can be either a cross-reference marker (a type 9 marker) or a unique string of text.

What the user sees

When you insert a cross-reference, you select a cross-reference format, which specifies the information provided by the cross-reference. For more information on cross-reference formats, see “Cross-reference formats” on page 127.

How the API represents cross-reference instances

The API represents each cross-reference instance with the following:

- `FTI_XRefBegin` and `FTI_XRefEnd` text items that specify the ID of the `FO_XRef` object
- An `FTI_String` text item, which provides the text that appears where the cross-reference is inserted
- An `FO_XRef` object

`FO_XRef` properties provide the following information:

- The ID of an `FO_XRefFmt` object
- The ID of the next `FO_XRef` object in the document
- The name of the file in which the cross-reference source is located
- The element ID of the cross-reference, if it is in a structured flow in a document

Suppose you insert the cross-reference shown in Figure 2-25, using the See Heading & Page cross-reference format shown in Figure 2-24.

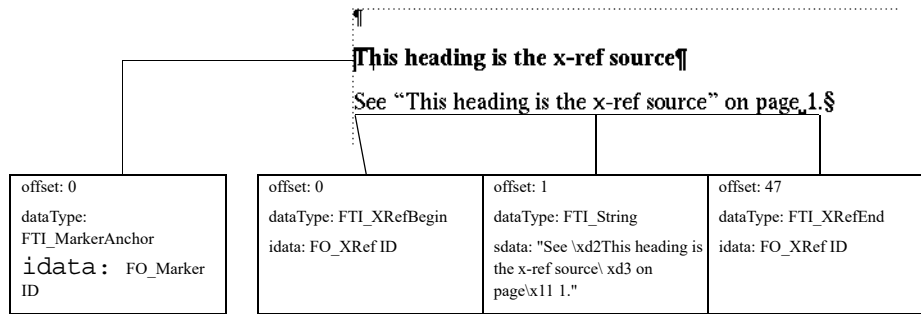


Figure 2-25 A cross-reference and the text items that represent it

The following are some properties of the `FO_XRef` object that represents the cross-reference.

Property	Type	Value
<code>FP_XRefFmt</code>	<code>F_ObjHandleT</code>	ID of the <code>FO_XRefFmt</code> object representing the See Heading & Page cross-reference format
<code>FP_NextXRefInDoc</code>	<code>F_ObjHandleT</code>	ID of the next <code>FO_XRef</code> object in document
<code>FP_XRefFile</code>	<code>StringT</code>	An empty string ("")

Client-owned cross references

A client can use the following properties of the `FO_XRef` object for identifying the cross-references it owns and to handle them specifically, as required:

- `FP_XRefClientName`
- `FP_XRefClientType`
- `FP_XRefSrcElemNonUniqueId`
- `FP_XRefAltText`.

A client can create its own dialog for cross-references. The notification `FA_Note_DisplayClientXRefDialog` is sent to the client to display or update (in case it is already displayed) this dialog. If the client displays or updates its cross-reference dialog, then it sets the return value as `FR_DisplayedXRefDialog` using the API, `F_ApiReturnValue()` to indicate this to FrameMaker. If the return value is not

set as explained, FrameMaker assumes that the client did not display any dialog. Consequently, FrameMaker's standard cross-reference dialog is displayed.

Variable formats

The user can insert variables that represent a variety of information, such as the page number or the date, into text. The information a variable provides is specified by a variable format.

What the user sees

Each variable format can specify a combination of text and building blocks. Building blocks are FrameMaker product-defined variables that you can string together.

There are six principal classes of variable formats:

- Page number
- Date
- Filename
- Table
- Running header or footer
- User

Each of these classes has a unique set of building blocks. You cannot use a building block from one class in another class. For example, you cannot use a date building block in a page number variable format.

How the API represents variable formats

FrameMaker products represent each variable format with an `FO_VarFmt` object, whose properties provide the following information:

- The name of the variable format
- The list of building blocks and text strings
- The type of variable it is (for example, page count or user variable)
- ID of the next `FO_VarFmt` object in the document

Suppose you edit the Creation Date (Long) variable format so that its definition is as shown in Figure 2-26.

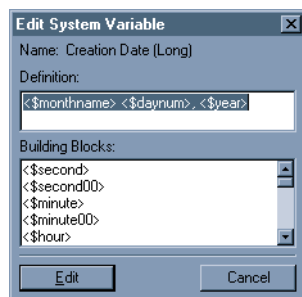


Figure 2-26 *Creation Date (Long) variable definition*

The following are some properties of the `FO_VarFmt` object that represents the Creation Date (Long) variable format.

Property	Type	Value
<code>FP_Fmt</code>	<code>StringT</code>	<code><\$monthname> <\$daynum>, <\$year></code>
<code>FP_SystemVar</code>	<code>IntT</code>	<code>FV_VAR_CREATION_DATE_LONG</code>
<code>FP_NextVarFmtInDoc</code>	<code>F_ObjHandleT</code>	ID of next <code>FO_VarFmt</code> object in the document

Variables

The user can insert variables in text. There are some restrictions on inserting some variable formats. For example, you can insert current page number, running header, and running footer variables only in an untagged flow on a master page.

What the user sees

The information an instance of a variable provides depends on its variable format. For example, if a variable's format is Page Count and the current document has 27 pages, each time the variable occurs in text, it appears as 27.

How the API represents instances of variables

The API represents each variable instance with the following:

- An `FTI_VarBegin` text item and an `FTI_VarEnd` text item that specify the ID of an `FO_Var` object
- An `FTI_String` text item that provides the text that appears where the variable is inserted
- An `FO_Var` object

`FO_Var` properties provide the following information:

- The ID of an `FO_VarFmt` object
- ID of the next `FO_Var` object in the document
- The element ID of the variable, if it is a structured variable in a document

Figure 2-27 shows an instance of the Creation Date (Long) variable and the text items that represent it.

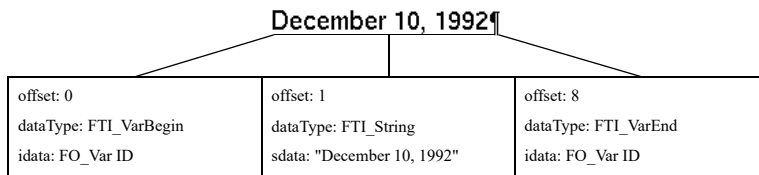


Figure 2-27 A variable instance and the text items that represent it

The following table lists some of the properties of the `FO_Var` object specified by `FTI_VarBegin` and `FTI_VarEnd` in Figure 2-27.

Property	Type	Value
<code>FP_VarFmt</code>	<code>F_ObjHandleT</code>	ID of the <code>FO_VarFmt</code> object that represents the Creation Date (Long) variable format
<code>FP_NextVarInDoc</code>	<code>F_ObjHandleT</code>	ID of the next <code>FO_Var</code> object in the document

Footnotes

A footnote is a type of special text column that appears at the bottom of a page.

What the user sees

A footnote reference (or anchor) appears in the main text as a number, letter, or special character. A footnote is visually separated from the main text by a separator (usually a horizontal line).

The Footnote Properties dialog box allows you to change characteristics that apply to all the footnotes in a document, such as the type of numbering or special characters used to represent the anchor and the height of the footnote column.

How the API represents footnotes

When the user chooses the Footnote command, the FrameMaker product inserts a footnote anchor. It also creates a text frame with invisible borders at the bottom of the text frame in which the footnote was inserted. The user types the footnote text into the footnote text frame.

Characteristics, such as the footnote anchor’s numbering type, are represented as document properties because they apply to all the footnotes in a document and not just individual footnote instances. For more information on the document properties that govern footnote characteristics, see “How the API represents documents” on page 76.

The API represents each footnote anchor with an `FTI_FnAnchor` text item, which specifies the ID of the `FO_Fn` object that represents the footnote. `FO_Fn` properties provide the following information:

- The footnote number
- The ID of the text frame in which the footnote text appears
- The ID of the next footnote in the list of footnotes in the document
- The IDs of the first and last paragraphs containing the footnote’s text
- The element ID of the footnote, if it is a structured footnote in a document

Figure 2-28 shows a footnote and the text item that represents it.

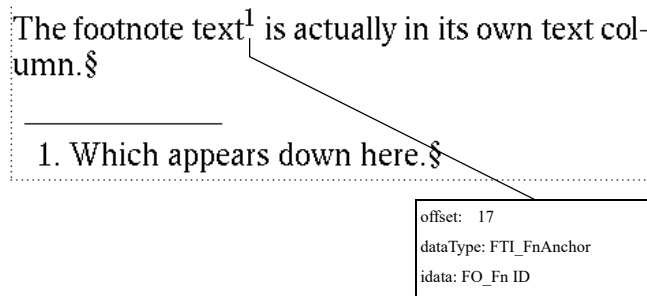


Figure 2-28 A footnote and the text item that represents it

The following table lists the properties of the `FO_Fn` object specified by the `FTI_FnAnchor` text item in Figure 2-28.

Property	Type	Value
<code>FP_InTextObj</code>	<code>F_ObjHandleT</code>	The ID of the subcolumn (<code>FO_SubCol</code>) in which the footnote appears.
<code>FP_FnNum</code>	<code>IntT</code>	0.
<code>FP_PrevFn</code>	<code>F_ObjHandleT</code>	ID of the previous footnote in the text frame (in this case, <code>NULL</code>).
<code>FP_NextFn</code>	<code>F_ObjHandleT</code>	ID of the next footnote in the text frame (in this case, <code>NULL</code>).
<code>FP_NextFnInDoc</code>	<code>F_ObjHandleT</code>	ID of the next footnote in the document.
<code>FP_FirstPgf</code>	<code>F_ObjHandleT</code>	ID of the first paragraph (<code>FO_Pgf</code>) in the footnote.
<code>FP_LastPgf</code>	<code>F_ObjHandleT</code>	ID of the last paragraph (<code>FO_Pgf</code>) in the footnote (in this example, it is the same as the one specified by <code>FP_FirstPgf</code>).

Although `FP_FnNum` specifies an integer, the number that appears in the document can be one of several ordinal or special characters. For example, if you set the document's `FP_FnNumStyle` property to `FV_FN_NUM_ALPHA_UC`, an *A* would appear instead of the *1* in the body text and at the beginning of the footnote.

To get all the paragraphs in a footnote, you traverse the `FP_NextPgfInFlow` and `FP_PrevPgfInFlow` properties, just as you would to get the paragraphs in any other flow.

Ruling Formats

Each document has a Ruling Catalog containing several ruling formats.

What the user sees

Rulings are the lines that border a table cell or an entire table. A ruling format specifies a line type (such as Thin or Thick) and the gap between the line and the cell contents.

You can specify rulings for an entire table in the Table Designer or for individual table cells in the Custom Ruling and Shading dialog box. FrameMaker products provide default rulings, such as Thick and Thin. You can change these rulings or create your own.

How the API represents ruling formats

The API represents a ruling format with an `FO_RulingFmt` object, whose properties provide the following information:

- The name of the ruling format
- Its line width
- The gap between lines if the ruling specifies double lines
- The ID of the next `FO_RulingFmt` object in the document

Suppose you create the Medium ruling format shown in Figure 2-29.

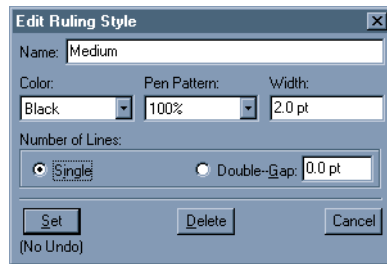


Figure 2-29 *Medium ruling format*

The API represents this ruling format with an `FO_RulingFmt` object. The following table lists some of its properties.

Property	Type	Value
<code>FP_Name</code>	<code>StringT</code>	Medium
<code>FP_RulingPenWidth</code>	<code>MetricT</code>	2*pts
<code>FP_RulingGap</code>	<code>MetricT</code>	0
<code>FP_RulingLines</code>	<code>IntT</code>	2

How the API organizes ruling formats

The API organizes the `FO_RulingFmt` objects in a document in a linked list. The `FO_Doc` property, `FP_FirstRulingFmtInDoc`, specifies the first `FO_RulingFmt` object in the list. Each `FO_RulingFmt` object has an `FP_NextRulingFmtInDoc` property, which specifies the ID of the next `FO_RulingFmt` object in the list. The order of the list does *not* correspond to the order in which the formats appear in the Custom Ruling and Shading dialog box.

Table Catalog formats

Each document has a Table Catalog containing table formats.

What the user sees

When you create a new table, you specify a format from the Table Catalog. The format provides the following information:

- The Table Catalog format name
- Format characteristics, such as the table position, alignment, and rulings
- The number of columns and rows

If you tag an existing table with a Table Catalog format, the Table Catalog format provides only the format name and the format characteristics for the table; it does not affect the number of columns or rows.

After you have created a new table or tagged an existing table, you can change the number of columns or rows or the format without affecting the Table Catalog tag. You can also instruct the FrameMaker product to apply the changes to the Table Catalog tag and other tables tagged with the format. You can modify or delete formats that are already in the Table Catalog, or you can add new formats.

How the API represents Table Catalog formats

FrameMaker products represent each Table Catalog format with an `FO_TblFmt` object, whose properties provide the following information:

- The name of the table format
- Format characteristics
- The default number of initial columns and rows
- The ID of the next `FO_TblFmt` object in the document

Suppose you create the Table Catalog format described in the Table Designer and the Insert Table dialog box in Figure 2-30.

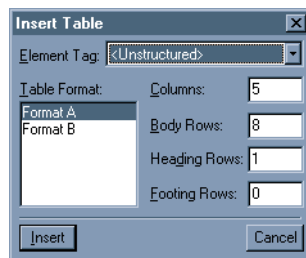
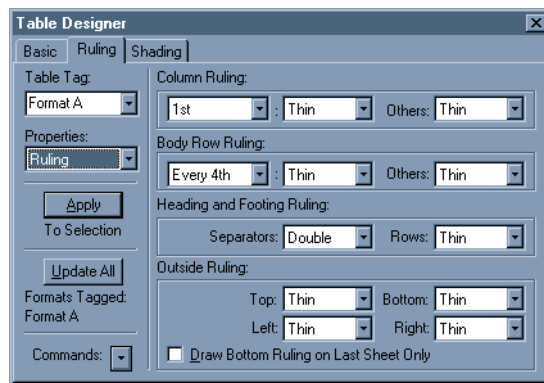


Figure 2-30 Table Catalog format

The following table lists some of the properties of the `FO_TblFmt` object that represents Format A.

Property	Type	Value
<code>FP_Name</code>	<code>StringT</code>	Format A
<code>FP_TblInitNumCols</code>	<code>IntT</code>	5
<code>FP_TblInitNumHRows</code>	<code>IntT</code>	1
<code>FP_TblColRuling</code>	<code>F_ObjHandleT</code>	ID of <code>FO_RulingFmt</code> object that represents the Thin line ruling
<code>FP_TblBodyRowRuling</code>	<code>F_ObjHandleT</code>	ID of <code>FO_RulingFmt</code> object that represents the Thin line ruling
<code>FP_TblHFSeparatorRuling</code>	<code>F_ObjHandleT</code>	ID of <code>FO_RulingFmt</code> object that represents the Double line ruling

How the API organizes Table Catalog formats

The API organizes the formats in the Table Catalog in a linked list.

The `FO_Doc` object property, `FP_FirstTblFmtInDoc`, specifies the ID of the first `FO_TblFmt` object in the list. Each `FO_TblFmt` object has an `FP_NextTblFmtInDoc` property, which specifies the ID of the next `FO_TblFmt` object in the list. The order of the list does *not* correspond to the order in which the formats appear in the Table Designer.

Tables

FrameMaker products allow you to insert tables into text. When you insert a table, a table anchor symbol (^) appears on the screen at the point where you inserted it.

What the user sees

Tables are useful for organizing information in cells arranged in rows and columns. Tables can have titles and heading, body, and footing rows. FrameMaker products automatically repeat table titles and heading and footing rows on each page of a table.

Each cell in a table is actually a type of text frame. It can contain text and nearly anything you insert in text, such as an anchored frame or a marker. You cannot insert another table directly into a table cell.

Like a paragraph, each table has a tag and a format. The tag is the name of a Table Catalog format. A table format specifies the layout characteristics of a table, such as its

position in a text frame, its alignment, and the rulings and shadings of its columns and rows. The table format can specify different rulings for the different types of rows (for example, the body, heading, and footing rows).

You can override a Table Catalog format by changing an individual table’s format. You can also override an individual table’s format by specifying a custom ruling, shading, or color for an individual cell or set of cells in the table. If you retag a table with a Table Catalog format after you have specified custom ruling for some of its cells, it does not affect the custom ruling for those cells.

How the API represents tables

The API represents a table anchor with an `FTI_TblAnchor` text item. For more information on text items, see “How the API represents text” on page 114. Each `FTI_TblAnchor` text item specifies the ID of an `FO_Tbl` object.

The API represents the table itself with the following objects:

- An `FO_Tbl` object
- One or more `FO_Row` objects
- One or more `FO_Cell` objects

FO_Tbl and table formats

`FO_Tbl` properties provide the following information:

- The table format tag (name)
- Formatting (such as alignment and rulings)
- The number of columns and rows
- The ID of the paragraph in the table’s title
- IDs of `FO_Row` objects that represent the first and last rows in the table
- ID of the next `FO_Tbl` object in the document
- The element IDs of the table, table title, table heading, table body, table footing elements if the table is a structured table in a document

`FO_Tbl` formatting properties are the same as `FO_TblFmt` formatting properties, except they do not include properties (such as `FP_TblInitNumHRows`) that specify the initial numbers of rows or columns.

The table title

If a table has a title, the `FO_Tbl` properties, `FP_FirstPg` and `FP_LastPg`, specify the IDs of the first and last `FO_Pgf` objects in the title.

Rows

The API represents each row in a table with an `FO_Row` object. `FO_Row` properties provide the following information about a table row:

- Its type (heading, body, or footing)
- Whether it is kept with the previous row, the next row, or both when a page break occurs within the table
- Its maximum and minimum allowable height
- The IDs of the `FO_Row` objects that represent the rows before and after it in the table
- The ID of the `FO_Cell` object that represents the first (leftmost) cell in the row
- The conditions that apply to the row
- The element ID of the row, if it is a structured row in a document

If a row has conditions applied to it, its `FP_InCond` property specifies an `F_IntsT` structure that includes the IDs of the `FO_CondFmt` objects that represent the conditions.

Cells

The API represents each cell in a table with an `FO_Cell` object, whose properties provide the following information:

- The ruling and shading that the cell inherits from the table format
- Custom ruling and shading
- Flags that indicate whether the cell's custom shading and fill override the table's shading and fill
- IDs of sibling `FO_Cell` objects
- IDs of the first and last paragraphs in the cell
- The element ID of the cell, if it is a structured cell in a FrameMaker document

The `FO_Cell` object properties, `FP_FirstPgf` and `FP_LastPgf`, specify the IDs of the first and last paragraphs in the cell. If there is more than one paragraph in the cell, each paragraph's `FP_PrevPgfInFlow` and `FP_NextPgfInFlow` properties specify the IDs of the paragraphs before and after it.

`FO_Cell` objects have two properties for each ruling. For example, the properties for the top ruling are `FP_CellDefaultTopRuling` and `FP_CellOverrideTopRuling`. The default ruling is the ruling that the cell inherits from the `FO_Tbl` object that contains it. For example, the `FP_CellDefaultTopRuling` property for a cell in a body row inherits the value of the `FP_TblBodyRowRuling` property in the `FO_Tbl` object that contains it. An

override ruling is a ruling that the user specifies in the Custom Ruling and Shading dialog box for an individual cell. If a value is specified for an override ruling, it overrides the default ruling.

The `FO_Cell` properties, `FP_CellOverrideShading` and `FP_CellOverrideFill`, specify the cell's custom shading and fill. If the cell's custom shading and fill override the table's default shading and fill, `FP_CellUseOverrideFill` and `FP_CellUseOverrideShading` are `True`.

How the API organizes the objects that represent tables

Figure 2-31 shows a table anchor, a table, and the text item that represents the anchor.

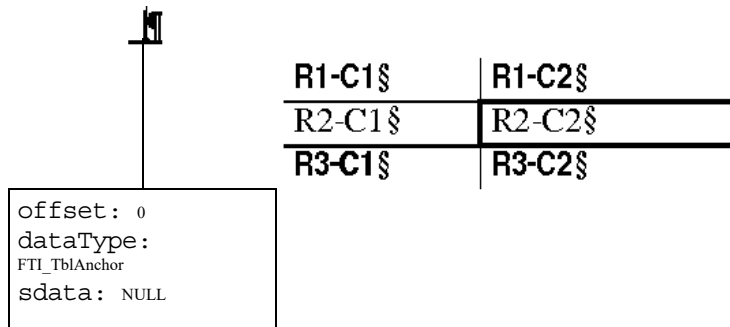


Figure 2-31 *A table and the text item that represents its anchor*

The API represents the table with the objects shown in Figure 2-32.

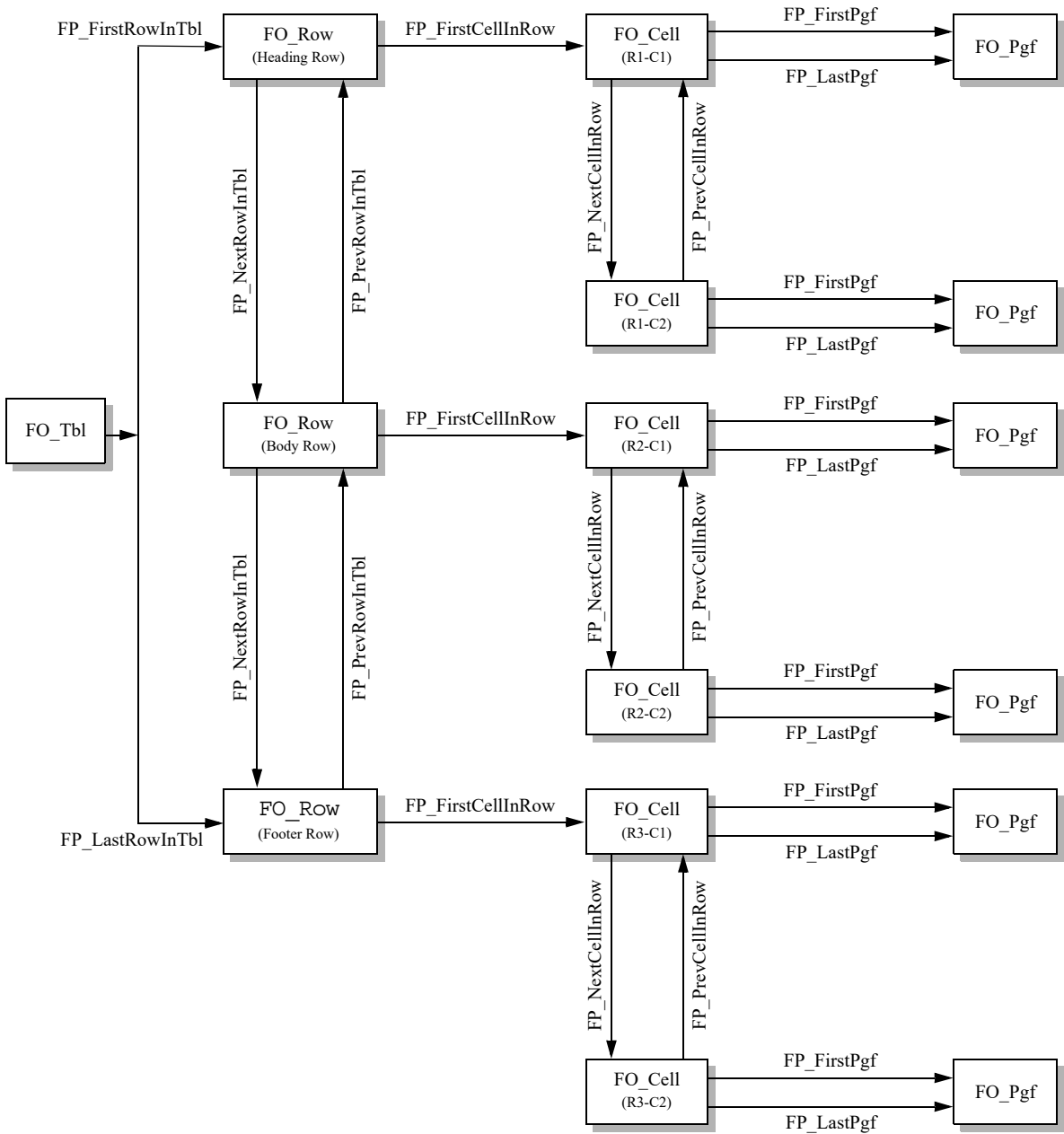


Figure 2-32 Objects that represent a table

The following table lists some of the `FO_Tbl` object's properties.

Property	Type	Value
<code>FP_TblTag</code>	<code>StringT</code>	Format A
<code>FP_TblTopRuling</code>	<code>F_ObjHandleT</code>	NULL
<code>FP_TblHFSeparatorRuling</code>	<code>F_ObjHandleT</code>	ID of <code>FO_RulingFmt</code> that represents Double line
<code>FP_TblBodyRowRuling</code>	<code>F_ObjHandleT</code>	ID of <code>FO_RulingFmt</code> that represents Thin line
<code>FP_TblBodyFirstFill</code>	<code>IntT</code>	0
<code>FP_FirstRowInTbl</code>	<code>F_ObjHandleT</code>	ID of <code>FO_Row</code> that represents the heading row (row 1)
<code>FP_LastRowInTbl</code>	<code>F_ObjHandleT</code>	ID of <code>FO_Row</code> that represents the footing row (row 3)

The following are some of the properties of the `FO_Row` object that represents the table's heading row.

Property	Type	Value
<code>FP_PrevRowInTbl</code>	<code>F_ObjHandleT</code>	NULL
<code>FP_NextRowInTbl</code>	<code>F_ObjHandleT</code>	ID of <code>FO_Row</code> that represents row 2
<code>FP_RowType</code>	<code>IntT</code>	<code>FV_ROW_HEADING</code>
<code>FP_RowKeepWithNext</code>	<code>IntT</code>	True
<code>FP_FirstCellInRow</code>	<code>F_ObjHandleT</code>	ID of <code>FO_Cell</code> that represents the R1-C1 cell

The following are some properties of the `FO_Cell` object that represents the R2-C2 cell. For the cell's override fill and rulings to override the fill and ruling provided by the table's format, the `FP_CellUseOverrideCharacteristic` properties must be set to `True`.

Property	Type	Value
<code>FP_CellOverrideFill</code>	<code>IntT</code>	5
<code>FP_CellUseOverrideFill</code>	<code>IntT</code>	<code>True</code>
<code>FA_CellDefaultLeftRuling</code>	<code>F_ObjHandleT</code>	ID of <code>FO_RulingFmt</code> that represents the Medium line ruling
<code>FP_CellOverrideLeftRuling</code>	<code>F_ObjHandleT</code>	ID of <code>FO_RulingFmt</code> that represents the Thick line ruling
<code>FP_CellDefaultBottomRuling</code>	<code>F_ObjHandleT</code>	ID of <code>FO_RulingFmt</code> that represents the Medium line ruling
<code>FP_CellOverrideBottomRuling</code>	<code>F_ObjHandleT</code>	ID of <code>FO_RulingFmt</code> that represents the Thick line ruling

How the API represents straddle table cells

When the user straddles a set of table cells, the FrameMaker product links all of the paragraphs in the cells. It changes the `FP_FirstPgf` and `FP_LastPgf` properties of the first cell (topmost and leftmost) so that it specifies the first and last paragraphs of the new linked list of paragraphs. All the other cells specify paragraph IDs of zero. It changes the properties of the first `FO_Cell` object in the straddle as listed in the following table.

Property	New value
<code>FP_CellIsStraddled</code>	False
<code>FP_CellNumRowsStraddled</code>	The number of rows in the straddle
<code>FP_CellNumColsStraddled</code>	The number of columns in the straddle

It also changes the properties of cells other than the first cell in the straddle as listed in the following table.

Property	New value
<code>FP_CellIsStraddled</code>	True
<code>FP_CellNumRowsStraddled</code>	1
<code>FP_CellNumColsStraddled</code>	1

The straddle uses the custom rulings and shadings of the first cell. When the user unstraddles the cells, the FrameMaker product leaves all the paragraphs that were in the straddle in the first cell. It gives each other cell a new empty paragraph. It leaves the original custom rulings and shadings of each cell intact.

Suppose you straddle both cells in a table row that has two cells. Each cell contains a single paragraph before you straddle them. Figure 2-33 shows how the `FO_Cell` objects appear before and after they are straddled and unstraddled. The FrameMaker

product automatically inserts a new paragraph in the second cell after you unstraddle the cells.

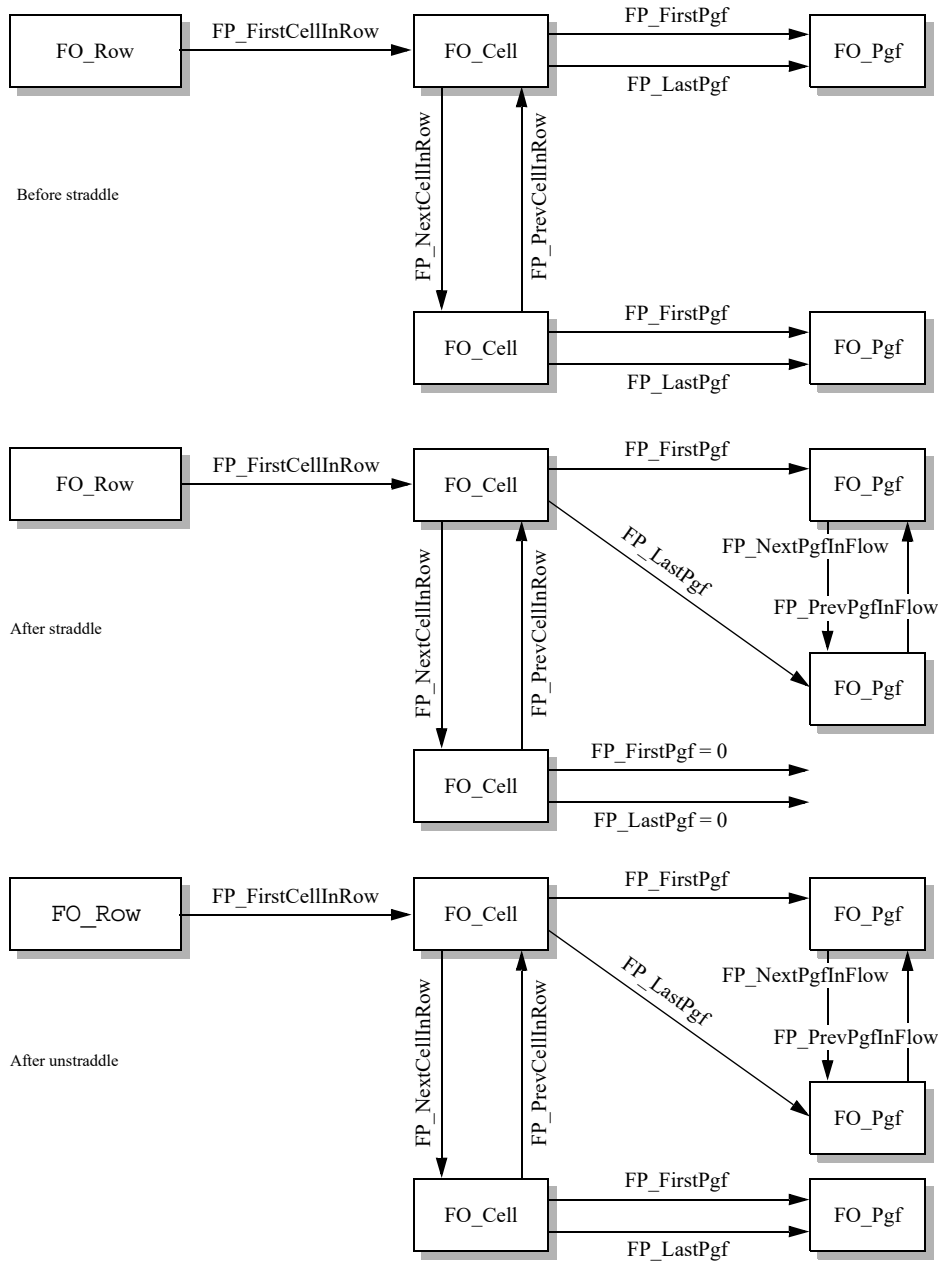


Figure 2-33 Table cells before and after straddle

Colors

You can assign spot colors to text and objects in a document, and you can print process color separations. You can also set up color views to specify which colors are visible in a document.

What the user sees

FrameMaker products provide a set of default colors. You can also define your own colors and store them in the document's Color Catalog. The FrameMaker product provides three color models for creating your own colors: CMYK, RGB, and HLS. It also provides eight color libraries; Crayon, DIC, FOCOLTONE, Greys, MUNSELL, PANTONE®, TOYO, and TRUMATCH.

How the API represents colors

The API represents each default color and each user-defined color with an `FO_Color` object. Tints are special `FO_Color` objects because each tint includes a reference to a base color, which is itself an `FO_Color` object. For a tint, the `FP_TintBaseColor` property returns the object ID of the base `FO_Color` object.

Paragraph formats, graphic objects, and other objects to which you can apply a color have properties that specify the ID of an `FO_Color` object. For example, all graphic objects have an `FP_Color` property that specifies the ID of an `FO_Color` object.

The properties of `FO_Color` objects provide the following information:

- The name of the color
- The color library and associated pigment used for the color
- The CMYK values of the color
- Tint base color and percentage
- Whether the color overprints or knocks out when printing

Library colors, `FP_FamilyName`, and `FP_InkName`

When you specify a color from a library, the `FP_Cyan`, `FP_Magenta`, `FP_Yellow`, and `FP_Black` properties are set to represent the library color. If you later set `FP_FamilyName` and `FP_InkName` to `NULL`, the CMYK settings remain the same, but the `FO_Color` object no longer defines a library color.

Both `FP_FamilyName` and `FP_InkName` are required to uniquely define a library color. The order in which you set the values of these properties is important. You must set a valid value for `FP_FamilyName` before you set `FP_InkName`. If you try to set the

ink name when the family name is set to NULL, `F_ApiSetString()` returns an error of `FE_NoColorFamily`.

When you set a value for `FP_FamilyName`, two things can happen:

- If the current value for `FP_InkName` specifies a valid ink for the newly set `FP_FamilyName`, then `FP_InkName` does not change.
- If the current value for `FP_InkName` does not specify a valid ink for the newly set `FP_FamilyName`, then the value for `FP_InkName` automatically changes to the first ink name for the new color family.

If you set `FP_FamilyName` to a color family that is not installed on your system, `F_ApiSetString()` returns an error of `FE_BadFamilyName`. If you set `FP_InkName` to a name that is not included in the current family, `F_ApiSetString()` returns an error of `FE_BadInkName`.

If you set one of either `FP_FamilyName` or `FP_InkName` to NULL, then the other property value automatically changes to NULL.

Formal color library names and ink names

Note that you must specify the family name as the as the formal color library name, including the registered trademark symbol. For example, the following sets the color library for a color to MUNSELL® Book of Color; note the code `(\xa8)` for the “®” character.

```
F_ApiSetString(docId, baseId, FP_FamilyName,
               "MUNSELL\xa8 Book of Color");
```

When specifying an ink name, you don’t necessarily provide the full ink name as described in the color library’s reference material. Some ink names have prefixes or suffixes that are not used by the API.

The following table lists the formal name for each color library that FrameMaker products support, along with an example of a legal string to specify an ink name via the FDK:

Color library name	Ink name
Crayon	Apricot
DIC COLOR GUIDE SPOT	2298p*
FOCOLTONE	1070
Greys	49% Grey.prcs
MUNSELL® High Chrome Colors	2.5R 7:10
MUNSELL® Book of Color	2.5R 9:1

Color library name	Ink name
PANTONE® Coated	Yellow 012
PANTONE® ProSim	Process Yellow
PANTONE® Uncoated	Yellow 012
PANTONE ProSim EURO®	Process Yellow
PANTONE® Process CSG	1-1
PANTONE® Process Euro	E 1-1
TOYO COLOR FINDER	0001pc*
TRUMATCH 4-Color Selector	1-a

Tinted colors

In FO_Color objects that are tints, the following properties have no meaning:

FP_FamilyName

FP_InkName

FP_Cyan

FP_Magenta

FP_Yellow

FP_Black

Changing these properties in a tinted color will turn the FO_Color object into an untinted color. If you want to change the hue of a tinted FO_Color object, you must select a new base color or change the hue of the base color.

Also, you cannot change FP_ColorPrintCtl and FP_ColorViewCtl in a tinted color; if you try to change them, the FDK returns an error of FE_TintedColor. To change these properties, you must change them in the tint's base color.

FP_TintPercent

You can set FP_TintPercent to a metric value from 0.00 to 100.0 (representing 0% to 100%), or to FV_COLOR_NOT_TINTED. If you set it to FV_COLOR_NOT_TINTED, then FP_TintBaseColor automatically changes to FV_NO_BASE_COLOR.

When you set a percentage value for FP_TintPercent, if FP_TintBaseColor was set to FV_NO_BASE_COLOR, then it automatically changes to the object ID for the color Black.

FP_TintBaseColor

Every tint has a base color. Note that you cannot use a tint as a base color for some other tint. If you set the base color to `FV_NO_BASE_COLOR`, then the `FP_TintPercent` for the current `FO_Color` object is set to `FV_COLOR_NOT_TINTED`.

When you set a valid color for `FP_TintBaseColor`, if `FP_TintPercent` was initially set to `FV_COLOR_NOT_TINTED`, then it will automatically be set to the metric value of `100.0` (for 100%). Be sure to change the tint percent if you want less than 100%.

Reserved colors

FrameMaker products have eight reserved colors. `FO_Color` objects have a read-only property named `FP_ReservedColor` to specify whether the object represents a reserved color or not. Unless the color is one of the eight reserved colors, `FP_ReservedColor` will always be `FV_COLOR_NOT_RESERVED`. `FP_ReservedColor` can have one of the following values:

`FV_COLOR_NOT_RESERVED`

`FV_COLOR_CYAN`

`FV_COLOR_MAGENTA`

`FV_COLOR_YELLOW`

`FV_COLOR_BLACK`

`FV_COLOR_WHITE`

`FV_COLOR_RED`

`FV_COLOR_GREEN`

`FV_COLOR_BLUE`

For a reserved color, all the properties are read-only except `FP_ColorOverPrint`, `FP_ColorPrintCtl`, and `FP_ColorViewCtl`. If you try to change any of the read-only properties, the FDK returns an error of `FE_ReservedColor` (except for properties that normally return `FE_ReadOnly` for unreserved colors).

Structural element definitions

A structured FrameMaker document has an Element Catalog, which contains structural element definitions and named format change lists.

There are two ways to test whether a document is structured via the API. To test whether a document contains structure elements, get the `FP_HighestLevelElement` property for the main `FO_Flow` object in the document. To test whether the document contains

an element catalog, get the `FP_FirstElementDefInDoc` property for the `FO_Doc` object. If you get legal values for these properties, then the document contains structure elements or an element catalog.

What the user sees

Each structural element definition has a name (tag), which usually corresponds to a type of document component or structural element, such as `Section`, `List`, `Quotation`, or `BodyPara`. An element definition specifies an element's relationship to other elements in a structured document. An element definition can also contain formatting information about the element.

The parts of an element definition that specify an element's format are known as the *format rules*.

The part of an element definition that specifies a container element's contents is known as a *content rule*. The content rule includes the following:

- A *general rule*, which specifies what elements are inside the container and in what order
- A list of *inclusions*, which specifies other elements that can appear anywhere in a container or the elements it includes (its *descendants*)
- A list of *exclusions*, which specifies elements that cannot appear in a container or in its descendants

Element definitions also specify *attribute definitions*, which describe attributes or separate units of information that the user can store with an element. An attribute definition can specify that an attribute is required for all elements with the element definition. It can also provide a list of the values an attribute can have, as well as a default value.

How the API represents structural element definitions

FrameMaker represents each element definition with an `FO_ElementDef` object. `FO_ElementDef` properties provide the following information about an element definition:

- Its name
- Its format rules
- Comments which describe its use
- Its attribute definitions
- Its content rule
- The type of element it defines (for example, a container or a system variable)

- Flags indicating whether the element definition is defined in the Element Catalog and whether it can be used as the highest-level element for a flow
- The ID of the next `FO_ElementDef` object in the document
- Initial structure rules for automatic insertion of child elements
- Initial pattern rules for table components

The API uses an `FO_FmtRule` object to represent each of an element definition's format rules. The `FO_ElementDef` object has the following properties that specify an element definition's format rules:

- `FP_FirstPgfRules`
- `FP_LastPgfRules`
- `FP_ObjectFmtRules`
- `FP_PrefixRules`
- `FP_SuffixRules`
- `FP_TextFmtRules`

Each of these properties specifies an `F_IntsT` structure, which provides a list of `FO_FmtRule` IDs.

For example, suppose you create the element definition shown in Figure 2-34.

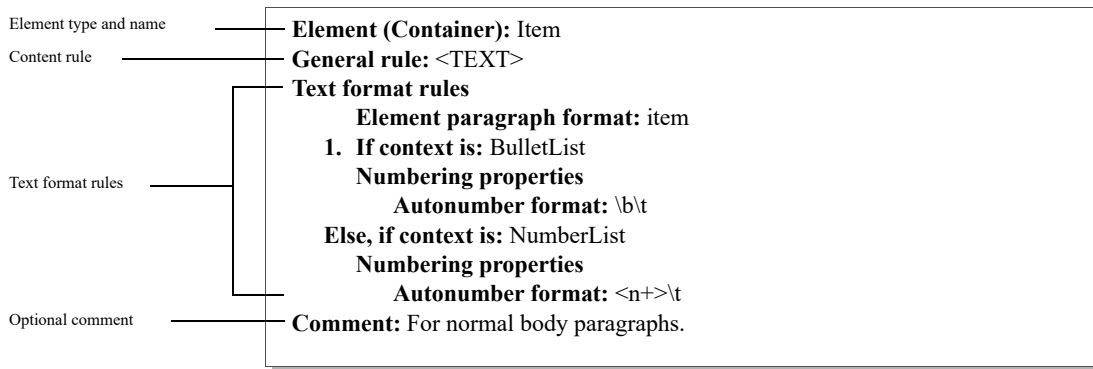


Figure 2-34 Element definition for Item element

The API represents the element definition in Figure 2-34 as an `FO_ElementDef` object with the following properties:

Property	Type	Value
<code>FP_Name</code>	<code>StringT</code>	Item.
<code>FP_ElementPgFormat</code>	<code>StringT</code>	item.
<code>FP_NextElementDefInDoc</code>	<code>F_ObjHandleT</code>	ID of the next <code>FO_ElementDef</code> object in the Element Catalog.
<code>FP_ElementInCatalog</code>	<code>IntT</code>	True.
<code>FP_GeneralRule</code>	<code>StringT</code>	<TEXT>.
<code>FP_ObjectType</code>	<code>IntT</code>	<code>FV_FO_CONTAINER</code> .
<code>FP_Comment</code>	<code>StringT</code>	For normal body paragraphs.
<code>FP_TextFmtRules</code>	<code>F_IntsT</code>	The ID of the element definition's text format rule. For a list of this format rule's properties, see "How the API represents format rules and format rule clauses" on page 156.

Format rules and format rule clauses

An element definition can contain several format rules, each of which can contain several format rule clauses.

What the user sees

Format rules and format rule clauses allow the template builder to specify the formats an element has in specific circumstances. A format rule can be either a *context rule* or a *level rule*.

A context rule contains clauses that specify an element's formatting based on its parent and sibling elements. For example, one clause of a format rule could specify that a `Para` element has the `FirstBody` paragraph format if it is the first child of a `Heading` element. Another clause could specify that a `Para` element has the `Body` paragraph format in all other contexts.

A level rule contains clauses that specify an element's formatting based on the level to which it is nested within specific types of ancestor elements. For example, one clause of

a level rule could specify that a `Para` element appears in 12-point type if has only one `Section` element among its ancestors. Another clause could specify that a `Para` element appears in 10 point type if there are two `Section` elements among its ancestors.

A format rule clause can use any of the following to specify an element's formatting in specific contexts:

- A formatting tag or name, such as a paragraph tag, a character tag, or a marker name
- A subformat rule
- A format change list
- A named format change list

How the API represents format rules and format rule clauses

The API uses an `FO_FmtRule` object to represent each format rule in an element definition, and an `FO_FmtRuleClause` object to represent each format rule clause in a format rule. Each `FO_FmtRule` object has an `FP_FmtRuleClauses` property, which specifies its format rule clause.

`FO_FmtRule` properties provide the following information about a format rule:

- A list of its format rule clauses
- An indication of whether it is a context rule or a level rule
- If the format rule is a level rule, the element tags to count among the element's ancestors and the tag at which to stop counting

`FO_FmtRuleClause` properties provide the following information about a format rule clause:

- A flag indicating how the rule clause specifies formatting, such as a paragraph tag or a format change list
- The formatting tag or name, subformat rule, or change list the rule clause uses to specify the element's formatting
- The circumstances under which the rule clause applies: if it is in a context rule, the context; if it is in a level rule, the level
- The context label

For example, the element definition shown in Figure 2-34 on page 154 includes a single format rule. The following table lists some of the properties of that format rule.

Property	Type	Value
FP_ElementDef	F_ObjHandleT	ID of the FO_ElementDef object that contains the format rule.
FP_FmtRuleClauses	F_IntsT	The IDs of the format rule clauses in the format rule. For a list of the rule clause's properties, see the table below.
FP_FmtRuleType	IntT	FV_CONTEXT_RULE.

The F_IntsT structure specified by the FO_FmtRule object's FP_FmtRuleClauses property provides an array, which includes the IDs of two format rule clauses. The following table lists some of the properties of the first format rule clause.

Property	Type	Value
FP_ContextLabel	StringT	BulletList.
FP_FmtChangeList	F_ObjHandleT	The ID of the format change list (FO_FmtChangeList object) that is applied to the element when the specified context is valid. For a list of the change list's properties, see "How the API represents format change lists" on page 158.
FP_FmtRule	F_ObjHandleT	ID of the FO_Rule object that contains the format rule clause.
FP_RuleClauseType	IntT	FV_RC_CHANGELIST.

Format change lists

A format change list describes a set of changes to paragraph format properties.

What the user sees

A format rule clause can use format change lists to specify how a paragraph format changes when the format rule clause applies. A change list can specify a change to just a single paragraph property, or it can specify changes to a long list of properties.

A change list can specify absolute values or relative values. For example, it can specify that the paragraph left indent is one inch, or it can specify that it is one inch greater than the inherited left indent.

A change list can be *named* or *unnamed*. A named change list appears in the Element Catalog. Format rule clauses that use a named change list specify its name (or tag). Multiple rule clauses can specify the same named change list. An unnamed change list appears in a rule clause. It is used only by the rule clause in which it appears.

How the API represents format change lists

The API uses an `FO_FmtChangeList` object to represent each change list in a document. `FO_FmtChangeList` properties provide the following information about a change list:

- Its name if it is a named change list
- The ID of the next change list in the document's list of change lists
- A paragraph format tag if the change list specifies one

A change list has one property for each paragraph format property it changes. For example, if it changes only the first indent, it has the properties described above and just an `FP_FirstIndent` property. If it changes the space below and the leading, it has the properties described above and the `FP_SpaceBelow` and `FP_Leading` properties.

If a change list changes a paragraph property to an absolute value, the property it uses has the same name as the corresponding paragraph format property (for example, `FP_FirstIndent`). If the change list changes a property with a relative value, the property it uses has the name of the corresponding paragraph format property with the word `Change` appended to it (for example, `FP_FirstIndentChange`).

For example, the format rule clause in the element definition in Figure 2-34 on page 154 includes an unnamed change list. The following table lists the change list's properties.

Property	Type	Value
<code>FP_Name</code>	<code>StringT</code>	<code>NULL</code>
<code>FP_NextFmtChangeListInDoc</code>	<code>F_ObjHandleT</code>	ID of the next <code>FO_FmtChangeList</code> object in the document
<code>FP_PgfCatalogReference</code>	<code>StringT</code>	<code>NULL</code>
<code>FP_AutoNumString</code>	<code>StringT</code>	<code>\b\t</code>

Structural elements

Structured Framemaker documents contain structural elements, which are instances of structural element definitions.

What the user sees

Each structural element is a component of a document. A structural element can consist of one or more paragraphs, a text range, one or more child elements, or anything you can insert in text (such as variables or tables).

Each structural element has an element definition specifying what its format and contents should be. Elements in a document can have the same element definition. For example, a document may have several elements with a Para element definition. For more information on element definitions, see “Structural element definitions” on page 152.

The elements in a container element are called its child elements. Child elements can also be containers; container elements can be nested. The element definition’s general rule specifies a container’s allowable child elements or text and the order in which they should occur.

A container element can violate its content rule by omitting required child elements, by including excluded child elements, or by having the elements in the wrong order. If a container element obeys its content rule, it is said to be valid.

Elements can also have attributes, which correspond to XML attributes. An attribute can be a *defined attribute*, which is defined in the element’s element definition, or an *undefined attribute*, which is not defined in the element’s element definition.

How the API represents structural elements

FrameMaker represents a structural element with an `FO_Element` object, whose properties provide the following information:

- Its attributes
- The ID of the object that represents its element definition
- The IDs of its parent and immediate sibling elements
- The IDs of the first and last `FO_Element` objects in the linked list of its child `FO_Element` objects
- Whether the element is collapsed
- Whether the element is valid; and if it is invalid, the reasons it is invalid

- The ID of the object associated with the `FO_Element` object, if the element is a noncontainer element, such as a marker or a system variable
- The element's context label
- The format rule clauses that apply to the element

How the API represents a structural element's validity

An element can be invalid in several ways. For example, its parent's content rule may not allow it, or it may contain a child element that is not allowed. `FO_Element` objects have validation properties that indicate the extent of an element's validity. For example, an element has a property named `FP_ElementIsValidInParent`, which is set to `True` if the element is not allowed by its parent element.

Changing an element

When using FrameMaker, an author can select elements and wrap, merge, or change them. The API provides functions to wrap and merge elements directly; for example, `F_ApiMergeIntoFirst()`. However, there is no corresponding function to directly change an element. Changing an element corresponds to a user selecting an element in the document, selecting an element name in the Element catalog, and then clicking Change on the element catalog.

To change an element via the API, you must change the `FP_ElementDef` property of the `FO_Element` object. You can traverse the list of element definitions in the document by starting with the `FP_FirstElementDefInDoc` property of the `FO_Doc` object, and then using the `FP_NextElementDefInDoc` property of the resulting `FO_ElementDef` object. You can identify the element definition by using its `FP_Name` property.

Frame Book Architecture

⋮

This chapter describes books and discusses how the Frame API represents them.

What the user sees

A book maintains a collection of documents that are known as *components*. The book helps you organize and format these component documents. It also enables you to create generated files, such as tables of contents and indexes. A book does *not* contain the component document files. It contains *references* to the component document files, in an ordered list; such a reference is called a book *component*.

Each component contains its own setup data such as pagination and numbering. For example, each component contains properties to determine whether its page and paragraph numbering continues from the previous document or restarts at 1, and whether the document starts on a left or right page.

The component properties should not differ from the corresponding properties in the document file. However, by setting the values in one but not both, a client could set up a component with different numbering properties than the corresponding document file. Subsequent book updates will make the numbering properties match.

For example, the paragraph numbering for a set of document in files might be set to restart at 1. When the user adds the documents to the book, then each component will have the same numbering properties. Your client could loop through a book and set the numbering for each component to continue from the previous file. In this case, when your client (or the user) updates the book, the FrameMaker product will change the numbering for the document files so they match the component numbering. For more information about how component and document numbering properties interact, see your Frame product user's manual.

.....
IMPORTANT: *A book component can be a document saved in any file format. The FDK can only modify document objects in documents that were saved in FrameMaker binary (FASL) files, but you can use channels to open MIF or text files and modify them.*
.....

How the API represents books

The API represents each book with an `FO_Book` object which can contain one or more `FO_BookComponent` objects.

`FO_Book` objects have properties that provide the following information:

- Whether the book has been modified
- Display properties such as book window size and location, text to show for each component, and text in the book's status line
- Whether or not the book is view-only, and view-only display properties
- Selection state; whether the book icon is selected, the first selected component, or the range of selected structure elements in the book
- Properties that determine how to print the book and save it as PDF
- For structured books, structure properties such as the element catalog for the book and the ID of the highest level element in the book

An `FO_BookComponent` object represents an individual book component. It has properties that provide the following information:

- The name of the document represented by the component
- The IDs of the next and previous component in the book and the next selected component in the book
- Whether the component is generated, and the type of generated file; in other words, whether the component is a specific type of list or index
- Whether to include the component in print, update, and import formats operations
- The list of paragraph format tags the product uses to generate a list from this component
- The ID of the parent book
- Numbering and pagination properties for the component; these properties may differ from the document's specific set of numbering properties
- For structured books, the structural element representing the book component

Suppose you create the book in Figure 3-1. The book is named `C:\MyDocs\book1.book`, and has three documents; `myDoc1.fm`, `myDoc2.fm`, and `myDoc3.fm`.

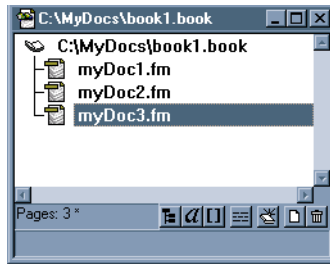


Figure 3-1 A book with three components

The API represents `mybook.book` with one `FO_Book` object and three `FO_BookComponent` objects. The following table lists some of the `FO_Book` object's properties.

Property	Type	Value
<code>FP_Name</code>	<code>StringT</code>	<code>C:\MyDocs\book1.book</code>
<code>FP_NextOpenBookInSession</code>	<code>F_ObjHandleT</code>	0
<code>FP_FirstComponentInBook</code>	<code>F_ObjHandleT</code>	ID of <code>FO_BookComponent</code> object for <code>myDoc1.fm</code>
<code>FP_FirstSelectedComponentInBook</code>	<code>F_ObjHandleT</code>	ID of <code>FO_BookComponent</code> object for <code>myDoc3.fm</code>
<code>FP_StatusLine</code>	<code>StringT</code>	Empty string (""); the status line currently displays no text
<code>FP_TypeOfDisplayText</code>	<code>IntT</code>	<code>FV_BK_FILENAME</code>

The following code shows how to get properties from the selected book component. First it gets the active book, and then the first selected component in the active book. For the book in Figure 3-1, the component would be for `myDoc3.fm`. For the selected component, the code prints out the method the component uses to compute footnote numbering. Then, if the footnote numbering uses custom characters for footnotes (daggers, etc.), the code prints out the custom numbering string.

```

VoidT F_ApiCommand(command)
    IntT command;
{
    F_ObjHandleT bookId, compId;
    StringT numString;

    bookId = F_ApiGetId(0, FV_SessionId, FP_ActiveBook);
    compId = F_ApiGetId(FV_SessionId, bookId,
        FP_FirstSelectedComponentInBook);

    F_Printf(NULL, "\n CompName is: %s",
        F_ApiGetString(bookId, compId, FP_Name));

    switch(F_ApiGetInt(bookId, compId, FP_FnNumComputeMethod)) {
        case FV_NUM_RESTART:
            F_Printf(NULL, "\nFn Compute: FV_NUM_RESTART");
            break;
        case FV_NUM_CONTINUE:
            F_Printf(NULL, "\nFn Compute: FV_NUM_CONTINUE");
            break;
        case FV_NUM_PERPAGE:
            F_Printf(NULL, "\nFn Compute: FV_NUM_PERPAGE");
            break;
        case FV_NUM_READ_FROM_FILE:
            F_Printf(NULL, "\nFn Compute: FV_NUM_READ_FROM_FILE");
            break;
        default:
            F_Printf(NULL, "\nFn Num Compute Method: UNKNOWN");
            break;
    }

    if(F_ApiGetInt(bookId, compId, FP_FnNumStyle)
        == FV_FN_NUM_CUSTOM) {
        F_Printf(NULL, "\nFn Num Style: FV_FN_NUM_CUSTOM");
        numString = F_ApiGetString(
            bookId, compId, FP_FnCustNumString);
        F_Printf(NULL, "\n    Cust Str: %s", numString);
        F_ApiDeallocateString(&numString);
    }
}

```

How the API organizes book components

The API organizes the `FO_BookComponent` objects that represent a book's components in a linked list. The `FO_Book` object's `FP_FirstComponentInBook` property specifies the first `FO_BookComponent` object in the list. Each `FO_BookComponent` object has `FP_PrevComponentInBook` and `FP_NextComponentInBook` properties that specify the IDs of the previous and next `FO_BookComponent` objects in the list. The order of the list is the same as the order of the components in the book.

Suppose you create the book shown in Figure 3-1 on page page 163. The API represents this book with the objects shown in Figure 3-2.

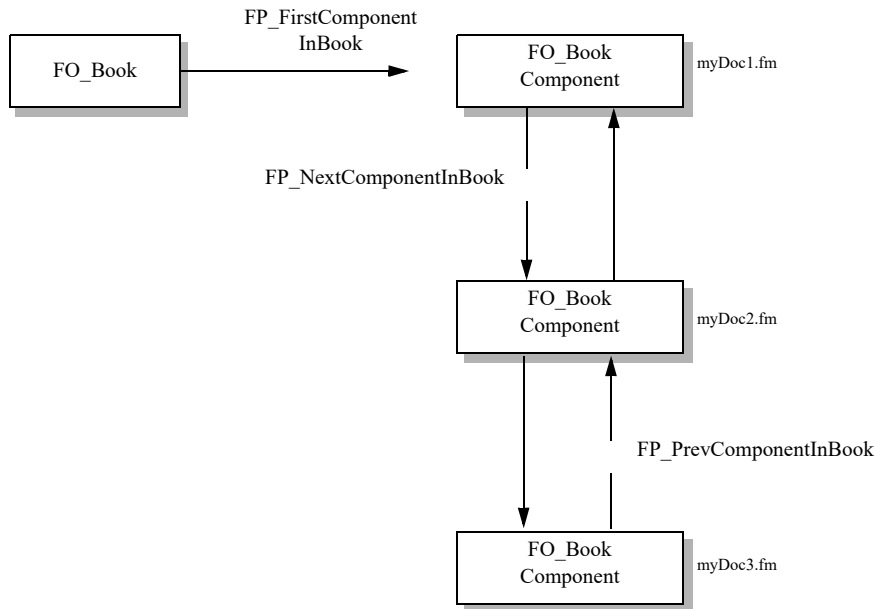


Figure 3-2 Objects that represent a book and its components

How the API represents structured books

If a book is a FrameMaker structured book, it has an `FP_HighestLevelElement` property, which specifies the ID of the `FO_Element` object that represents the root element. Each component in the book also has an `FP_ComponentElement` property, which specifies the `FO_Element` object ID for that component's structure element.

Creating new books and components

To create a new book, use `F_ApiNewNamedObject()`. If you pass an empty string for the object name, the FrameMaker product creates an untitled book.

To insert a new book component in a book, use `F_ApiNewSeriesObject()`. This creates a new book component, but the component has no document file assigned to it. If the user tries to open such a component's file, the FrameMaker product will alert the user that no file exists to match the component name. When you insert a new book component, you should assign a document to it.

For more information on `F_ApiNewSeriesObject()`, see "Creating series objects" on page 369.

The following code creates an untitled book with one component that represents a newly created, custom document file. After the code successfully saves the new document, it creates a book component and assigns the document name to the new component.

```
#define in ((MetricT) 65536*72)
. . .
VoidT F_ApiCommand(command)
    IntT command;
{
    F_ObjHandleT bookId, docId, compId;
    StringT s;

    F_ApiBailOut();
    bookId = F_ApiNewNamedObject(FV_SessionId,
                                FO_Book, (StringT) "");
    /* First create an 8.5 x 11 custom document. */
    docId = F_ApiCustomDoc(F_MetricFractMul(in,17,2), 11*in, 1,
                          F_MetricFractMul(in,1,4), in, in, in, in,
                          FF_Custom_SingleSided, True);
    /* Save the doc, prompting for filename */
    F_ApiSimpleSave(docId, (StringT) "", True);
    /* If file has name, make a component */
    s = F_ApiGetString(FV_SessionId, docId, FP_Name);
    if(F_StrLen(s)) {
        compId = F_ApiNewSeriesObject(
                bookId, FO_BookComponent, 0);
        F_ApiSetString(bookId, compId, FP_Name, s);
        F_ApiDeallocateString(&s);
    }
}
```

Updating a book

After making changes in the documents or components of a book, your client must update the book to ensure all references are valid. For example, if you change the page numbering properties, you must update the book to ensure cross-references indicate the correct numbering. You can update a book via the `F_ApiUpdateBook()` command or the `F_ApiSimpleGenerate()` command.

With `F_ApiUpdateBook()`, you can specify aspects of the Update operation, such as whether to update books with view-only documents. You can specify all aspects of the operation, or you can specify some aspects and allow the user to decide others. For example, you can instruct the FrameMaker product to post an alert if the book contains a MIF file.

To use `F_ApiUpdateBook()`, you should first understand property lists and how to manipulate them directly. For more information on this subject, see “Representing object characteristics with properties” on page 65 and “Manipulating property lists directly” on page 295.

The syntax for `F_ApiUpdateBook()` is:

```
ErrorT F_ApiUpdateBook(F_ObjHandleT bookId,
    F_PropValsT *updateParamsp,
    F_PropValsT **updateReturnParamssp);
```

This argument	Means
<code>docId</code>	The ID of the document or book to save.
<code>updateParamsp</code>	A property list that tells the FrameMaker product how to update the book and how to respond to errors and other conditions. Use <code>F_ApiGetUpdateBookDefaultParams()</code> or <code>F_ApiAllocatePropVals()</code> to create and allocate memory for this property list. To use the default list, specify <code>NULL</code> .
<code>updateReturnParamssp</code>	A property list that returns information about how the FrameMaker product updated the book.

.....
IMPORTANT: *Always initialize the pointer to the property list that you specify for `updateReturnParamssp` to `NULL` before you call `F_ApiUpdateBook()`.*

To call `F_ApiUpdateBook()`, do the following:

- 1 Initialize the pointer to the `updateReturnParamssp` property list to `NULL`.

- 2 *Create an `updateParams` property list.*
You can get a default list by calling `F_ApiGetUpdateBookDefaultParams()`, or you can create a list from scratch.
- 3 *Call `F_ApiUpdateBook()`.*
- 4 *Check the Update status.*
Check the returned values in the `updateReturnParams` list for information about how the FrameMaker product updated the book.
- 5 *Deallocate the `updateParams` and `updateReturnParams` property lists.*
Steps 2, 4, and 5 are discussed in the following sections.

Creating an `updateParams` script with `F_ApiGetUpdateBookDefaultParams()`

The API provides a function named `F_ApiGetUpdateBookDefaultParams()` that creates a default `updateParams` property list. If you are setting a number of properties, it is easiest to use this function get the default property list and then change individual properties as needed.

The syntax for `F_ApiGetUpdateBookDefaultParams()` is:

```
F_PropValsT F_ApiGetUpdateBookDefaultParams();
```

The following table lists some of the properties in the property list returned by `F_ApiGetUpdateBookDefaultParams()`. The first value listed for each property is the default value for the property. You can change any property in the list to use its other legal values.

Property	Meaning and possible values
<code>FS_AlertUserAboutFailure</code>	Specifies whether to notify the user if something unusual occurs during the update operation False: don't notify user True: notify user
<code>FS_MakeVisible</code>	Make newly generated files (lists and indexes) visible True: make visible False: don't make visible

Property	Meaning and possible values
FS_ShowBookErrorLog	<p>Specifies whether to use the book error log to display warnings.</p> <hr/> <p>False: don't display book error log; display warnings in the console</p> <hr/> <p>True: display the book error log</p>

For the complete list returned by `F_ApiGetUpdateBookDefaultParams()`, see “`F_ApiGetUpdateBookDefaultParams()`” in the FDK Programmer’s Reference guide.

For example, to get a default `updateParamsp` property list and modify it so that it instructs `F_ApiUpdate()` to show the book error log, use the following code:

```

. . .
F_PropValsT params;
ErrorT err;
. . .

/* Get the default parameter list. */
params = F_ApiGetUpdateBookDefaultParams();
/* Get the index for the error log property, */
/* then set the property to True. */
i = F_ApiGetPropIndex(&params, FS_ShowBookErrorLog);
params.val[i].propVal.u.ival = True;
. . .

```

The API allocates memory for the property list created by `F_ApiGetUpdateBookDefaultParams()`. Use `F_ApiDeallocatePropVals()` to free the property list when you are done with it.

Creating an `updateParamsp` script from scratch

If you want to specify only a few properties when you call `F_ApiUpdateBook()`, it is most efficient to create a property list from scratch. To create the property list, you must allocate memory for it, and then set up the individual properties.

Use the API convenience function, `F_ApiAllocatePropVals()`, to allocate memory for the property list. For example, the following code creates an

updateParamsp property list that will instruct `F_ApiUpdateBook()` to display the error log:

```
#DEFINE ERR_LOG 0

. . .
F_PropValsT params, *returnParamsp = NULL;
. . .
/* Allocate memory for the list. */
params = F_ApiAllocatePropVals(1);
/* Set up FS_ShowBookErrorLog property and set it to True. */
params.val[ERR_LOG].propIdent.num = FS_ShowBookErrorLog;
params.val[ERR_LOG].propVal.valType = FT_Integer;
params.val[ERR_LOG].propVal.u.ival = True;
. . .
/* When you're finished, free the F_PropValsT */
F_ApiDeallocatePropVals(&params)
```

Checking update status

`F_ApiUpdateBook()` stores a pointer to a property list in `updateReturnParamsp`; the list contains one property which contains flags to indicate the status. For a list of the possible flags, see “`F_ApiUpdateBook()`” in the FDK Programmer’s Reference guide.

To determine if a particular `FS_UpdateBookStatus` bit is set, use `F_ApiCheckStatus()`. For example, the following code determines if an Update operation was canceled because the current book contains duplicate files (components that refer to the same file):

```
. . .
F_PropValsT params, *returnParamsp = NULL;
F_ObjHandleT bookId;

/* Get the ID of the active book. */
bookId = F_ApiGetId(0, FV_SessionId, FP_ActiveBook);

params = F_ApiGetUpdateBookDefaultParams();

F_ApiUpdate(bookId, &params, &returnParamsp);

if (F_ApiCheckStatus(returnParamsp, FV_DuplicateFileInBook))
    F_ApiAlert("Duplicate files in book.",
              FF_ALERT_CONTINUE_NOTE);

/* Deallocate property lists. */
F_ApiDeallocatePropVals(&params);
F_ApiDeallocatePropVals(returnParamsp);
. . .
```

The API provides a utility function named `F_ApiPrintUpdateStatus()`, which prints the save error values to the console platforms. For more information, see “`F_ApiPrintUpdateBookStatus()`” in the FDK Programmer’s Reference guide.

Example

The following code updates the currently active book. The update operation will display the error log for any error conditions, will allow inconsistent numbering properties, and (since the code allows inconsistent numbering) will not update the numbering in the book. It then prints out the update status. Finally, the code deallocates the property lists that it used to update the book.

```

. . .
#include "futils.h"

IntT i;
UCharT msg[1024];
F_PropValsT params, *returnParamsp = NULL;
F_ObjHandleT bookId;

params = F_ApiGetUpdateBookDefaultParams();
i = F_ApiGetPropIndex(&params, FS_ShowBookErrorLog);
params.val[i].propVal.u.ival = True;
i = F_ApiGetPropIndex(&params, FS_AllowInconsistentNumProps);
params.val[i].propVal.u.ival = FV_DoOk;
i = F_ApiGetPropIndex(&params, FS_UpdateBookNumbering);
params.val[i].propVal.u.ival = False;

err = F_ApiUpdateBook(bookId, &params, &returnp);
F_ApiPrintUpdateBookStatus(returnp);

F_ApiDeallocatePropVals(&params);
F_ApiDeallocatePropVals(returnp);

. . .

```

Using the book error log

When updating a book, the FrameMaker product posts errors to a book error log. The error log is a FrameMaker document that lists error conditions and includes hypertext links to offending locations in the book's document files.

By default, FDK clients post book errors to the console. However, your clients can post errors to the log, and can include hypertext links in those messages.

Displaying the error log for book updates

By default, the FrameMaker product displays update errors in the console. You direct the FrameMaker product to display the error log via the property list you pass to `F_ApiUpdateBook()`. In that list, set the `FS_ShowBookErrorLog` flag to `True`. For more information about the update book properties, see “Creating an updateParamsp script with `F_ApiGetUpdateBookDefaultParams()`” on page 168.

Writing messages to the error log

The FrameMaker product includes an API client that writes messages to the error log. To write a message to the error log, you must use `F_ApiCallClient()`.

The syntax for the client call is:

```
F_ApiCallClient ("BookErrorLog",  
                "log -b=[bookId] -d=[docId] -o=[objId] -- [text]");
```

where:

- `BookErrorLog` is the name of the client to call.
- `log` identifies this as a log message.
- `-b` is either the book ID or a document ID; typically the active book.
- `-d` is either a document ID or an object ID; typically a document associated with a book component.
- `-o` is an object in the document represented by the `-d` argument. If you pass both a document ID and an object ID, the call adds a hypertext link, from the error message to the object you specified.
- `--` is the text of the message to appear in the log. To post a time stamp in the message, pass the `FM_PRINT_DATESTAMP` token as the message string.

The call creates a unique log for each book or document ID you pass in the `-b` argument; if you pass 0 for a book ID, you will create a log that is not associated with any book; all calls with the 0 book ID will go to that log file.

When you pass a document ID for the `-d` argument, the call creates an entry with the document's pathname. It then indents all contiguous entries with the same document ID under that document's pathname. This continues until you pass a different document ID. If you pass 0 for the `-d` argument, the call will not indent the errors.

If you don't have the document ID, you can specify log entry indenting under a filename via the text you pass for the log message. To do this, you precede the log message with a filename, followed by a carriage return. This method creates an indented section each time you pass a filename and carriage return, even if you pass the same filename in a series of log entries.

For example, if you passed the following to the BookErrorLog client in two consecutive calls:

```
"filename.fm\012Here is my first Log Message"
"filename.fm\012Here is my second Log Message"
The BookErrorLog client would create the following messages:
filename.fm
    Here is my first Log Message
filename.fm
    Here is my second Log Message
```

Example

The following code shows a function that posts messages to a log, with or without a time stamp; if you pass valid ID's for all the ID arguments, the log message will include a hypertext link to the specified object in the specified document:

```
VoidT ReportError(F_ObjHandleT docId, F_ObjHandleT objId,
                  ConStringT errmsg, BoolT dateStamp)
{
    F_ObjHandleT bookId;
    StringT log_msg = F_StrNew((UIntT)256);

    bookId = F_ApiGetId(0, FV_SessionId, FP_ActiveBook);
    if(dateStamp) {
        F_Sprintf(log_msg, "log -b=%d -d=%d -o=%d --%s",
                  bookId, docId, objId, (StringT)"FM_PRINT_DATESTAMP");
    } else {
        F_Sprintf(log_msg, "log -b=%d -d=%d -o=%d --%s",
                  bookId, docId, objId, errmsg);
    }
    F_ApiCallClient("BookErrorLog", log_msg);
    F_ApiDeallocateString(&log_msg);
}
```

PART III



Frame Application Program Interface

Introduction to the Frame API

.....

.....

This chapter provides an overview of how the API works and how to create an FDK client. It also provides a simple example—a client that you can create and run right away.

The API enables you to create a client that takes control of a FrameMaker product session. With the API, a client can do almost anything an interactive user can do. It can create, save, and print documents; add and delete text and graphics; and perform many other formatting and document-management tasks. It can also interact with the user by responding to user actions, displaying dialog boxes, and creating menus.

How the API works

The API represents everything in a FrameMaker product session as an *object*.¹ Each object has a *type*, a constant that indicates the type of thing it represents. For example, an object's type can be `FO_Doc` (if it represents a document), `FO_Rectangle` (if it represents a graphic rectangle), or `FO_Pgf` (if it represents a paragraph).

FrameMaker products assign an *identifier (ID)* to each object in a session. You use this ID to identify an object when you call API functions.

An object's characteristics are called *properties*. Each type of object has a particular set of properties or a *property list*. For example, an `FO_Rectangle` object's property list includes properties named `FP_Width` and `FP_Height`, which represent its height and width. An `FO_Pgf` object's property list includes properties named `FP_LeftIndent` and `FP_Leading`, which represent its left indent and its leading. Each property has a predetermined data type, such as `IntT` (integer, Boolean, or ordinal), `StringT` (string), or `F_ObjHandleT` (object ID).

Each of an individual object's properties has a *value*. This value describes the property for that particular object. For example, suppose a document contains a smoothed

.....

1. Frame API objects should not be confused with the graphic objects that you create with the Tools palette, object elements in structured documents, or the objects of object-oriented programming languages.

rectangle that is 20 points wide and 10 points high. The Frame API represents the rectangle as an `FO_Rectangle` object with the following properties and values.

Property	Data Type	Value
<code>FP_Width</code>	<code>MetricT</code>	20 * 65536 ^a
<code>FP_Height</code>	<code>MetricT</code>	10 * 65536
<code>FP_RectangleIsSmoothed</code>	<code>IntT</code>	True
<code>FP_FrameParent</code>	<code>F_ObjHandleT</code>	ID of the frame containing the rectangle

a. `MetricT` values are 32-bit integers that represent measurements in points. The 16 most significant bits represent the digits before the decimal. The 16 least significant bits represent the digits after the decimal. A point is 65536 ($1 \ll 16$) in `MetricT` units. For more information on `MetricT`, see “MetricT values” in the FDK Programmer’s Reference guide.

`FO_Rectangle` objects actually have many more properties than are shown in the table above. For a complete list of Frame API objects and properties, see chapter, “Object Reference,” in the FDK Programmer’s Reference guide.

How clients can change FrameMaker product documents, books, and sessions

A client can change FrameMaker documents, books, and sessions by:

- Creating and destroying objects
The API provides functions to create and destroy objects.
- Changing object properties
The API provides functions to get and set object properties.
- Programmatically executing FrameMaker product commands, such as Open, Print, Save, and Clear All Change Bars

How clients communicate with the user

A client can communicate with the user by:

- Creating menus and menu items
- Displaying dialog boxes

The API allows a client to respond to user actions by:

- Notifying the client when the user initiates certain events, such as Open, Save, or Quit
- Passing a message to the client when the user clicks a hypertext marker that contains a message *apiclient* hypertext command

How clients work with FrameMaker

Clients are dynamic link libraries (DLLs), or they can be executable programs that use COM to communicate with a FrameMaker session.. A client does not need to be aware of the low-level details of integrating with FrameMaker, because the API provides high-level functions that are the same on all platforms.

When the user starts FrameMaker, it sends an initialization call to each registered client. Clients can take control immediately, or they can request FrameMaker to notify them of specific events and wait for those events to occur.

Special types of clients

In addition to conventional clients that take control of a FrameMaker product session in response to user actions, the API allows you to create three special types of clients: document reports, filters, and take-control clients.

Document reports

A *document report* is a client that provides detailed information about a document. The user can start a document report by choosing Utilities>Document Reports from the File menu, and then choosing the report from the Document Reports dialog box. The FDK includes a sample document report, named `wordcnt`, which counts the number of words in a document.

Filters

A *filter* is a client that converts FrameMaker product files to or from other file formats.

An *import filter* is a filter that the FrameMaker product calls when the user attempts to open a non-Frame file and chooses a filter in the Unknown File Type dialog box. The

import filter reads the file and converts it to a FrameMaker product document or book. The FDK includes a sample import filter, named `mmlimport`, that converts MML files to FrameMaker product documents.

An *export filter* is a filter that the FrameMaker product calls when the user attempts to save a FrameMaker product document or book in a particular format by choosing the format in the Save dialog box or by specifying a filename with a particular extension. The export filter writes information in the document or book to a file with a different format.

A *file-to-file* filter is a filter that the FrameMaker product can call to both import or export files of different formats. A single file-to-file filter client can actually consist of more than one filter. For example, the same client could filter CGM to FrameVector and FrameVector to CGM. The way you register the client's different filters determines which filter to invoke for import and export. Another advantage of these filters is they can filter from an external file to an external file. For example, you could filter from CGM to TIFF, and the TIFF file can remain an external file that is imported into the document by reference.

Take-control clients

A *take-control client* is a client that takes control of a FrameMaker product session immediately after the FrameMaker product starts. Take-control clients are useful for conducting batch operations in which little or no user interaction is needed.

Portability

The API's design makes it easy to create portable clients. In most cases, you only need to recompile your client to run it on a different platform. To ensure that your client is completely portable, use the FDE with the API. You should replace platform-specific I/O, string, and memory functions in your client with the alternatives the FDE provides. For more information on the FDE, see Part III, "Frame Development Environment (FDE)."

The FDE and the API provide alternatives to the C language's fundamental data types. For example, the FDE and the API substitute `IntT` for a 32-bit `int` and `UCharT` for `unsigned char`. The API uses other types for specific purposes. For example, it uses `F_ObjHandleT` for object IDs. For a list of API data types, see chapter, "Data Types and Structures Reference," in the FDK Programmer's Reference guide.

Running clients with different FrameMaker product interfaces

FrameMaker ships with two product interfaces, Structured FrameMaker and FrameMaker. A client can only use functionality that is available in the product interface that is active for the currently running FrameMaker process. For example, if a client is running on the unstructured FrameMaker product interface, it can't create or manipulate structural elements (`FO_Element` objects). On the other hand, all functions in the FDK are available to a client running on the Structured FrameMaker product interface.

To determine if a function is available in a particular product interface, see chapter, "FDK Function Reference," in the FDK Programmer's Reference guide. For an example of how to programmatically determine which product interface is running, see "F_ApiGetString()" in the FDK Programmer's Reference guide.

Creating and running a client

To create and run a client, follow these general steps:

1 Write an initialization function.

Most clients need to define an `F_ApiInitialize()` callback function. When the FrameMaker product starts, it calls your client's `F_ApiInitialize()` function. Normally, you will want to include code in `F_ApiInitialize()` to set up your client's menus and request notification for particular events.

For more information on creating an `F_ApiInitialize()` callback, see Chapter 2, "API Client Initialization."

2 Set up the client's user interface.

Your client probably needs to interact with the user. To respond to user actions, you can define the following callback functions in your client:

- `F_ApiNotify()` to respond to the user initiating FrameMaker product operations, such as Open and Save
- `F_ApiCommand()` to respond to the user choosing menu items created by your client
- `F_ApiMessage()` to respond to the user clicking hypertext markers that contain the message `apiclient` command
- `F_ApiDialogEvent()` to respond to the user manipulating items in a dialog box created by your client

You can also display notices and prompt the user for input by using API dialog box functions, such as `F_ApiAlert()` and `F_ApiPromptString()`.

For more information on setting up your client’s user interface, see Chapter 3, “Creating Your Client’s User Interface.”

3 *Add code to programmatically execute FrameMaker product commands.*

Your client probably needs to execute some FrameMaker product commands, such as Open, Print, or Close. To execute these commands programmatically, use API functions, such as `F_ApiSimpleOpen()`, `F_ApiSilentPrintDoc()`, or `F_ApiClose()`.

For more information on using API functions to execute FrameMaker product commands, see Chapter 4, “Executing Commands with API Functions.”

4 *Add code to get and set object properties.*

To get or set an object property, use the `F_ApiGetPropertyType()` or `F_ApiSetPropertyType()` function that corresponds to the type of property you want to get or set. For example, to get or set an `IntT` property, use `F_ApiGetInt()` or `F_ApiSetInt()`. To get or set a `StringT` property, use `F_ApiGetString()` or `F_ApiSetString()`.

For more information on changing object properties, see Chapter 5, “Getting and Setting Properties.”

5 *Add code to create objects.*

To create objects, use the `F_ApiNewObjectType()` function that corresponds to the kind of object that you want to create. For example, to create a new anchored frame, use `F_ApiNewAnchoredObject()`.

For more information on creating objects, see Chapter 8, “Creating and Deleting API Objects.”

6 *Compile your client.*

The API function declarations are contained in the `fapi.h` header. Be sure to include this header in your client code. Include C library header files before the `fapi.h` header.

The FDK comes with sample makefiles or project files for each supported platform. To compile your client, use your platform’s make or build utility. For more information on using FDK makefiles or project files on a specific platform, see the *FDK Platform Guide* for that platform.

7 *Register your client with the FrameMaker product.*

The FrameMaker product needs to know about your client to initialize it. To let the FrameMaker product know about your client, you must make some changes to the environment under which the client runs.

- Add the following lines to the `[APIClients]` section of the `maker.ini` file:
`ClientName=ClientType, description, path, mode`
`ClientName` is the name that the FrameMaker product and other clients use to reference your client. `ClientType` specifies your client type: for example,

Standard, DocReport, or TextImport. *description* is a string describing your client. *path* is the pathname of your client's DLL. *mode* determines what product interfaces your client supports—can be one of all, maker, or structured.

You can also register a Windows client by setting values in the DLL's VERSIONINFO resource, then copying or moving the DLL in the FrameMaker product's Plugins folder..

A simple example

The following client adds a menu with three items to the FrameMaker product menu bar when the FrameMaker product starts. The first menu item closes the active document; the second item sets the fill pattern of a selected graphic object; the third item adds a body page to the active document. Following the code is a line-by-line description of how it works.

```

1 #include "fapi.h"
2 #define CloseDoc 1
3 #define SetFill 2
4 #define AddPage 3
5
6 VoidT F_ApiInitialize(initialization)
7 IntT initialization;          /* Code for initialization type */
8 {
9   F_ObjHandleT menuBarId, menuId;
10
11  /* Get ID of the FrameMaker product menu bar. */
12  menuBarId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
13                                 "!MakerMainMenu");
14  /* Add menu named "API" to the FrameMaker product menu bar. */
15  menuId = F_ApiDefineAndAddMenu(menuBarId, "APIMenu", "API");
16
17  /* Add items to API menu. */
18  F_ApiDefineAndAddCommand(CloseDoc, menuId, "CloseDocCmd",
19                           "Close", "\\!CD");
20  F_ApiDefineAndAddCommand(SetFill, menuId, "SetFillCmd",
21                           "Set Fill", "\\!SF");
22  F_ApiDefineAndAddCommand(AddPage, menuId, "AddPageCmd",
23                           "Add Page", "\\!AP");
24 }
25
26 VoidT F_ApiCommand(command)
27 IntT command;
28 {
29   F_ObjHandleT pgId, objId, docId;
30
```

```

31  /* Get the ID of the active document. */
32  docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
33  if(!docId) return; /* Return if there is no active document. */
34
35  switch (command)
36  {
37      case CloseDoc:      /* Close document even if it's changed. */
38          F_ApiClose(docId, FF_CLOSE_MODIFIED);
39          break;
40
41      case SetFill:      /* Set fill pattern to black. */
42          /* Get ID of selected object. */
43          objId = F_ApiGetId(FV_SessionId, docId,
44                          FP_FirstSelectedGraphicInDoc);
45
46          /* If an object is selected, set its fill. */
47          if (!objId) F_ApiAlert("Select an object first.",
48                              FF_ALERT_CONTINUE_WARN);
49          else F_ApiSetInt(docId, objId, FP_Fill, FV_FILL_BLACK);
50          break;
51
52      case AddPage:      /* Add a new page. */
53          pgId = F_ApiNewSeriesObject(docId, FO_BodyPage, 0);
54          break;
55  }
56  }

```

Lines 1–5

These lines include header files and define the constants for the client's menu items. All clients must include the `fapi.h` header. To ensure your client is portable, include `fapi.h` before any other header files.

Lines 6–25

These lines define the `F_ApiInitialize()` callback function. When the user starts the FrameMaker product, it calls `F_ApiInitialize()`.

The call to `F_ApiGetNamedObject()` gets the ID of the FrameMaker product menu bar (`!MakerMainMenu`). The call to `F_ApiDefineAndAddMenu()` creates a menu named API on the menu bar. The calls to `F_ApiDefineAndAddCommand()` add menu items to the API menu and define keyboard shortcuts for the items.

Lines 26–56

These lines define the `F_ApiCommand()` callback function. When the user chooses a menu item, the FrameMaker product calls this callback with `command` set to the menu item number (in this example, `CloseDoc`, `SetFill`, or `AddPage`).

Lines 31–34

These lines get the ID of the document to change. To use most API functions, you need to specify an ID. This example gets the ID of the document that has input focus, that is, the *active* document.

In each FrameMaker product session there is only one active document at a time. The session object (FO_Session) has a property named FP_ActiveDoc that specifies its ID. To retrieve the active document's ID from the FO_Session object, you use F_ApiGetId() to query the FO_Session object's FP_ActiveDoc property. The syntax for F_ApiGetId() is:

```
F_ObjHandleT F_ApiGetId(parentId, /* Object's parent */  
                        objId, /* Object whose property you want to query */  
                        property); /* Constant specifying property to query */
```

The `parentId` parameter specifies the ID of the object's parent—the session, book, or document that contains the object. No other object contains the FO_Session object, so `parentId` is set to 0. The ID of the FO_Session object (there can only be one) is always FV_SessionId, so `objId` is set to FV_SessionId.

Lines 37–40

These lines close the active document when the user chooses Close Doc from the API menu. The FF_CLOSE_MODIFIED flag instructs the API to close the document without warning the user, even if the document has unsaved changes.

Lines 41–51

These lines set the fill pattern of a selected object to black when the user chooses Set Fill from the API menu. To set the selected object's fill pattern, the client needs the object's ID. To get the ID, the client uses F_ApiGetId() to query the document property, FP_FirstSelectedGraphicInDoc. If no object is selected, F_ApiGetId() returns 0.

The F_ApiAlert() call displays an alert that tells the user to select an object. The constant, FF_ALERT_CONTINUE_WARN, specifies the type of alert—an alert with a Continue button.

To set the object's fill pattern, the client must set its `FP_Fill` property. `FP_Fill` is an `IntT` property, so the client must use `F_ApiSetInt()` to set it. The syntax for `F_ApiSetInt()` is:

```
VoidT F_ApiSetInt(parentId, /* Object's parent */
                  objId, /* Object whose property you want to set */
                  property, /* Constant specifying property to set */
                  value); /* Value to which to set the property */
```

`FP_Fill` can have any value between 0 and 15. The API-defined constant, `FV_FILL_BLACK`, specifies 0 (black).

Lines 52–54

These lines add a body page to the document when the user chooses Add a Page from the API menu. A body page object is a *series* object. To create a series object, you use `F_ApiNewSeriesObject()`. The syntax for `F_ApiNewSeriesObject()` is:

```
F_ObjHandleT F_ApiNewSeriesObject(parentId, /* Object's Parent */
                                   objectType, /* Constant specifying new object type */
                                   prevObjectId); /* Object for new object to follow */
```

The `parentId` parameter specifies the ID of the object that is to contain the new object. The new page should appear in the active document, so `parentId` is set to `docId`. The API uses `FO_BodyPage` objects to represent body pages, so `objectType` is set to `FO_BodyPage`. Specifying 0 for `prevObjectId` puts the new page at the beginning of the document. For more information on creating different types of objects, see “Creating objects” on page 361.

Compiling and running the example client

The source code for the example client and a makefile or project file are provided in provided in the `samples/myapi` directory of your FDK installation.

To compile the client, use your platform's make or build utility.

To run the example client, you must first register it. Assuming you have compiled your client into a DLL named `myapi.dll` and copied or moved it to the FrameMaker `fminit` directory, add the following line to the `maker.ini` file:

```
myapi = Standard,Ch. 1 Sample,fminit\myapi.dll
```

After you have registered the example client, start FrameMaker and open a document. The API menu should appear to the right of the FrameMaker menus.

Using old clients with FDK 12

For legacy clients compiled with FDK 10 to run successfully with FDK 12, they should be recompiled with Microsoft Visual Studio 2010.

API Client Initialization

.....

.....

This chapter describes how to start interaction between your client and FrameMaker.

Responding to the FrameMaker product's initialization call

When the FrameMaker product starts, it attempts to start all the clients registered with it,¹ except document reports and filters. The FrameMaker product attempts to start each client by calling its `F_ApiInitialize()` callback function.

Your client should define `F_ApiInitialize()` as follows:

```
VoidT F_ApiInitialize(initialization)
IntT initialization;
{
    /* Your client code goes here */
}
```

This argument	Means
<code>initialization</code>	A flag that indicates the type of initialization (see “Initialization types”)

Usually, you want your client to do something immediately after the user starts the FrameMaker product. For example, you may want to add menus to the menu bar or request notification for certain events. To do this, you call API functions from the `F_ApiInitialize()` function. For information on creating menus and requesting notification, see Chapter 3, “Creating Your Client’s User Interface.”

.....

1. For information on registering your client with a FrameMaker product, see the *FDK Platform Guide* for your platform.

Suppose you want your client to display a dialog box after the FrameMaker product is started. To do this, you could use the following `F_ApiInitialize()` function:

```

. . .
VoidT F_ApiInitialize(initialization)
IntT initialization;
{
    F_ApiAlert("Client has started.", FF_ALERT_CONTINUE_NOTE);
}
. . .

```

Initialization types

The following table summarizes the different types of initializations and the `initialization` constants FrameMaker products can pass to your client's `F_ApiInitialize()` callback.

Type of initialization	When <code>F_ApiInitialize</code> is called	Initialization constant	Clients that receive initialization
FrameMaker product starts with no special options	After starting	<code>FA_Init_First</code>	All except document reports and filters
FrameMaker product starts with take-control client	After starting	<code>FA_Init_First</code>	All except document reports and filters
	After all clients have finished processing the <code>FA_Init_First</code> initialization	<code>FA_Init_TakeControl</code>	All clients set up as take-control clients
Document report chosen from Document Reports dialog box	After report is chosen	<code>FA_Init_DocReport</code>	The chosen document report
Notification, menu choice, or hypertext command for a client that has bailed out	When the menu item is chosen, the hypertext command is clicked, or the notification should be issued	<code>FA_Init_Subsequent</code>	Clients that have bailed out and are waiting for an event, menu choice, or hypertext command to occur

First initialization

When the user starts a FrameMaker product, the FrameMaker product calls the `F_ApiInitialize()` function of each registered client (unless it's a document report or filter) with `initialization` set to `FA_Init_First`.

Take-control initialization

The FDK allows you to set up clients to receive a special initialization called a *take-control* or `FA_Init_TakeControl` initialization. The FrameMaker product issues the `FA_Init_TakeControl` initialization after it has issued the `FA_Init_First` initialization and all clients have returned control. This initialization is useful if you want your client to conduct some batch processing after other clients have initialized, but before the interactive user has control.

The FrameMaker product can issue the `FA_Init_TakeControl` initialization to several clients. To set up a client to receive `FA_Init_TakeControl` initializations, set the client's type to `TakeControl` in the FrameMaker product `.ini` file.

Document report initialization

When a FrameMaker product is started, it does not attempt to initialize API clients that are registered as document reports. It initializes a document report only when the user chooses the document report from the Document Reports dialog box. When this occurs, the FrameMaker product calls the document report's `F_ApiInitialize()` callback with `initialization` set to `FA_Init_DocReport`.

To appear in the Document Reports dialog box, a document report must be registered with the FrameMaker product as a document report. For information on registering document reports, see the *FDK Platform Guide* for your platform.

Filter initialization

If your client is registered as a filter, you should not define an `F_ApiInitialize()` function for it. When the user opens or saves a file and selects your filter, the FrameMaker product notifies your client by calling your client's `F_ApiNotify()` callback. For more information on `F_ApiNotify()` and notification, see “Responding to user-initiated events or FrameMaker product operations” on page 219.

To receive notification, your filter must be registered as a filter. For information on registering filters, see the *FDK Platform Guide* for your platform.

Initialization after a client has bailed out

If your API client is waiting for an event and not performing any other processing, it can call `F_ApiBailOut()`. This exits your client's process and frees all the system resources that it uses. If an event that your client is waiting for occurs, the FrameMaker product restarts your client by calling its `F_ApiInitialize()` function with `initialization` set to `FA_Init_Subsequent`.

A document report should always bail out after it completes processing, because the API initializes it each time the user chooses it from the Document Reports dialog box. A filter should always bail out after it filters a file, because the API initializes it each time a filterable file is opened, imported, or saved.

For more information on `F_ApiBailOut()`, see “`F_ApiBailOut()`” in the FDK Programmer's Reference guide.

Disabling the API

The user can disable all API clients before starting the FrameMaker product by changing the `API=On` setting in the FrameMaker product `.ini` file to `API=Off`.

.....
IMPORTANT: *Many FrameMaker features are implemented via API clients. If you disable the API then you also disable these features. Such features include XML and SGML import and export, Save As HTML, and Word Count.*

FrameMaker Product Activation by Asynchronous Clients

Asynchronous clients on Windows that launch a FrameMaker process and wait for it to become idle (by calling `WaitForInputIdle`) before attempting to connect using `F_ApiWinConnectSession` need to make provision for the connection time required during activation as explained below.

If FrameMaker has not been activated (relevant only for the FrameMaker Point Product on Windows XP and Windows Vista®), an activation screen prompts the user for input. The `WaitForInputIdle` call returns at this point while FrameMaker isn't actually ready for communication. Therefore, the client must give the user enough time to activate or skip activation before attempting to connect to FrameMaker using `F_ApiWinConnectSession`. Otherwise, the client can require the user to activate the product before using it. Despite activation, `WaitForInputIdle` returns too early before FrameMaker is actually ready to establish a connection. The issue can be resolved by modifying the code and introducing a 5-10 second sleep before attempting to connect to the FrameMaker session.

Another solution is to attempt to connect a multiple or indefinite number of times with short sleeps in between.

Asynchronous clients running the FrameMaker Point Product on Windows 2000 or running the FrameMaker Server on Windows 2000, XP, or VISTA won't encounter any such problems.

Creating Your Client's User Interface

.....

3

.....
:
:
:
:

This chapter describes how to use the Frame API to create a user interface for your FDK client.

Your client can interact with the user in the following ways:

- By displaying its own dialog boxes
- By implementing its own menus, menu items, and keyboard shortcuts
- By responding to the message *apiclient* hypertext command
- By responding to user-initiated events or FrameMaker product operations
- By implementing quick-key commands

The following sections discuss these ways of interacting with the user in greater detail.

Using API dialog boxes to prompt the user for input

The Frame API provides a variety of premade dialog boxes. All of these dialog boxes are *modal*—the user must dismiss them before continuing. The following sections discuss how to use these dialog boxes.

The API also allows you to create and use custom modal and modeless dialog boxes. For more information, see Chapter 10, “Creating Custom Dialog Boxes for Your Client” and Chapter 11, “Handling Custom Dialog Box Events.”

Using alert boxes

To display a dialog box with a short message, use `F_ApiAlert()`.

The syntax for `F_ApiAlert()` is:

```
IntT F_ApiAlert(StringT message,
                IntT type);
```

This argument	Means
<code>message</code>	The message that appears in the alert box
<code>type</code>	The type of alert box

Specify one of the following values for the `type` argument.

type constant	Type of dialog box displayed
<code>FF_ALERT_OK_DEFAULT</code>	Displays OK and Cancel buttons; OK is the default
<code>FF_ALERT_CANCEL_DEFAULT</code>	Displays OK and Cancel buttons; Cancel is the default
<code>FF_ALERT_CONTINUE_NOTE</code>	Displays Continue button
<code>FF_ALERT_CONTINUE_WARN</code>	Displays Continue button with a warning indication
<code>FF_ALERT_YES_DEFAULT</code>	Displays Yes and No buttons; Yes is the default
<code>FF_ALERT_NO_DEFAULT</code>	Displays Yes and No buttons; No is the default

`F_ApiAlert()` returns 0 if the user clicks OK, Continue, or Yes; otherwise, it returns a nonzero value.

Example

The following code displays the alert box shown in Figure 3-1:

```
...  
IntT err;  
  
err = F_ApiAlert((StringT)"This alert is an OK_DEFAULT.",  
                FF_ALERT_OK_DEFAULT);  
...
```

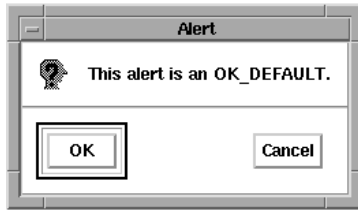


Figure 3-1 *FF_ALERT_OK_DEFAULT* alert box

Using string, integer, and metric input dialog boxes

To prompt the user for a single string, integer, or metric value, use `F_ApiPromptString()`, `F_ApiPromptInt()`, or `F_ApiPromptMetric()`. These functions all allow you to provide a default value for the entry field.

Their syntax is:

```
IntT F_ApiPromptString(StringT *stringp,  
                      StringT message,  
                      StringT stuffVal);
```

```
IntT F_ApiPromptInt(IntT *intp,  
                  StringT message,  
                  StringT stuffVal);
```

```
IntT F_ApiPromptMetric(MetricT *metricp,  
                      StringT message,  
                      StringT stuffVal,  
                      MetricT defaultunit);
```

This argument	Means
<code>stringp</code> , <code>intp</code> , or <code>metricp</code>	A pointer to the user variable that gets the return value from the input field when the user clicks OK.
<code>message</code>	The message that appears in the dialog box.

This argument	Means
<code>stuffVal</code>	The default value that appears in the input field when the dialog box is first displayed. It must be a string for integer and metric prompts, as well as string prompts.
<code>defaultunit</code>	The metric unit to use if the user doesn't specify one. For example, to use inches as the default unit, specify 4718592. For more information on metric values, see "MetricT values" in the FDK Programmer's Reference guide.

These functions all return 0 if the user clicks OK. Otherwise, they return a nonzero error value. If the user clicks Cancel, the API does not assign a value to `*stringp`, `*intp`, or `*metricp`.

If the user types alphabetic text after a number in an `F_ApiPromptInt()` dialog box, the API ignores the text and just returns the number. For example, if the user types 10 cookies, the returned value is 10.

`F_ApiPromptMetric()` dialog boxes behave like metric dialog boxes in the user interface. If the user types a number followed by a string that represents a unit, the API converts the value into the equivalent number of metric units. For example, if the user types 5in or 5", the API returns $5 * (4718592)$. If the user doesn't specify a unit, the API uses the unit specified by `defaultunit`.

.....
IMPORTANT: `F_ApiPromptString()` allocates memory for the string referenced by `*stringp`. Use the FDK function `F_ApiDeallocateString()` to free the string when you are done with it.

Examples

The following code displays the dialog box shown in Figure 3-2:

```
...
#include "fmemory.h"
IntT err;
StringT sres;
err = F_ApiPromptString(&sres, (StringT)"String?",
                       (StringT)"Default text");
if (err) return;
/* Some code to use the string goes here. */
F_ApiDeallocateString(&sres);
...
```

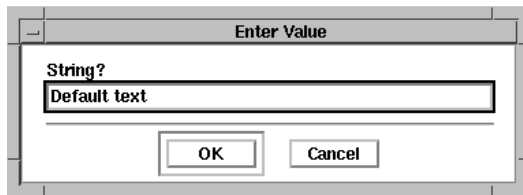


Figure 3-2 String prompt dialog box

The following code displays the dialog box shown in Figure 3-3:

```
...
#include "futils.h" /* Provides declaration for F_Sprintf(). */
IntT err, ires;
UCharT msg[256];

err = F_ApiPromptInt(&ires, (StringT)"Integer?", "1234");
if (err) F_Sprintf(msg,(StringT)"Cancelled, ires has no value");
    else F_Sprintf(msg,(StringT)"The value of ires is %d.",ires);
F_ApiAlert(msg, FF_ALERT_CONTINUE_NOTE);
...
```

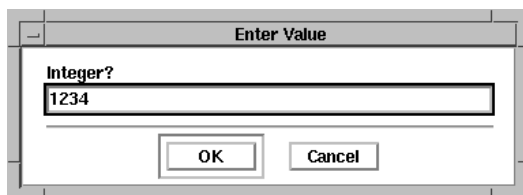


Figure 3-3 Integer prompt dialog box

The following code displays the dialog box shown in Figure 3-4:

```
. . .
#define IN (MetricT) 65536*72 /* Default unit (inches) */
IntT err;
MetricT mres;
err = F_ApiPromptMetric(&mres, (StringT)"Metric?",
                       "12.34in", IN);
. . .
```

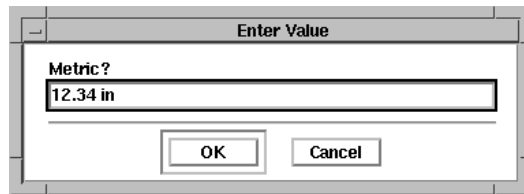


Figure 3-4 Metric prompt dialog box

Using file selection dialog boxes

To display dialog boxes similar to theFrameMaker product's Open and Save dialog boxes, use `F_ApiChooseFile()`. `F_ApiChooseFile()` displays files and directories in a scrolling list and allows the user to choose a file or directory.

The syntax for `F_ApiChooseFile()` is:

```
IntT F_ApiChooseFile(StringT *choice,
                    StringT title,
                    StringT directory,
                    StringT stuffVal,
                    IntT mode,
                    StringT helpLink);
```


This argument	Means
<code>choice</code>	The selected pathname when the user clicks OK.
<code>title</code>	The message that appears in the dialog box.
<code>directory</code>	The default directory when the dialog box is first displayed. If you specify an empty string, the last directory used by your client is used. If your client hasn't used any directories, the directory specified by the session property, <code>FP_OpenDir</code> , is used.
<code>stuffVal</code>	The default value that appears in the input field when the dialog box first appears. If the dialog box type specified by <code>mode</code> doesn't have an input field, this string is ignored.
<code>mode</code>	A constant specifying the type of dialog box. For a list of dialog box types, see "F_ApiChooseFile()" in the FDK Programmer's Reference guide.
<code>helpLink</code>	Obsolete in versions 6.0 and later; pass an empty string. The name of a document containing help information for the dialog box and an optional hypertext link.

.....
IMPORTANT: `F_ApiChooseFile()` allocates memory for the string referenced by `*choice`. Use `F_ApiDeallocateString()` to free the string when you are done with it.
.....

Example

To create the dialog box shown in Figure 3-5, add the following code to your client:

```

. . .
#include "futils.h"
#include "fmemory.h"

IntT err;
StringT sres;
UCharT msg[256];

err = F_ApiChooseFile(&sres, (StringT)"Choose a file",
                    (StringT)"/tmp", (StringT)"",
                    FV_ChooseSelect, (StringT)");
if (err)
    F_Printf(msg, (StringT)"Cancelled, sres is not defined.");
else
    F_Printf(msg, (StringT)"The value of sres is %s.", sres);

F_ApiAlert(msg, FF_ALERT_CONTINUE_NOTE);
if (!err) F_ApiDeallocateString(&sres);
. . .

```

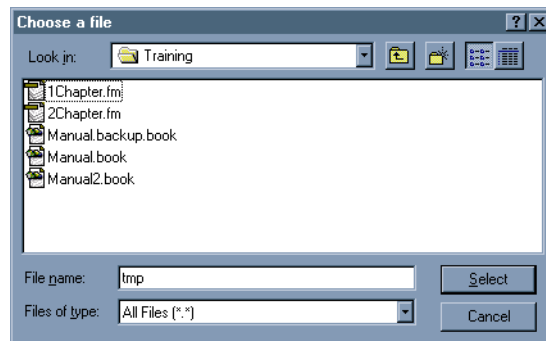


Figure 3-5 File selection dialog box

Using scroll list dialog boxes

To display a scroll list dialog box, use `F_ApiScrollBox()`. `F_ApiScrollBox()` displays an array of items that you provide and allows the user to choose one.

The syntax for `F_ApiScrollBox()` is:

```
IntT F_ApiScrollBox(IntT *selected_item,  
    StringT title,  
    F_StringsT *stringslist,  
    IntT default);
```

This argument	Means
<code>selected_item</code>	The index of the selected item when the user clicks OK (or double-clicks an item). The index of the first item is 0.
<code>title</code>	The title that appears on the dialog box.
<code>stringslist</code>	The list of items that appears in the scroll list.
<code>default</code>	The index of the item that is selected when the dialog box first appears. For no default, specify -1.

`F_StringsT` is defined as:

```
typedef struct {  
    UIntT len; /* Number of strings */  
    StringT *val; /* Array of strings */  
} F_StringsT;
```

The `F_ApiScrollBox()` function returns a nonzero value if the user clicks Cancel or an error occurs, or 0 if the user clicks OK.

Example

To create the dialog box shown in Figure 3-6, add the following code to your client:

```
. . .
#include "futils.h"
IntT err, choice, listLen = 3;
UCharT msg[256];
F_StringsT colors;
colors.val = (StringT *) F_Alloc(listLen*sizeof(StringT),
NO_DSE);
if (colors.val) {
    colors.len = (UIntT)listLen;
    colors.val[0] = F_StrCopyString("red");
    colors.val[1] = F_StrCopyString("green");
    colors.val[2] = F_StrCopyString("blue");

    err = F_ApiScrollBar(&choice, "Choose a color.", &colors,
0);

    if (!err)
        F_Printf(msg, "The choice is %s.", colors.val[choice]);
    else
        F_Printf(msg, "Cancel was pressed");
    F_ApiAlert(msg, FF_ALERT_CONTINUE_NOTE);
    F_ApiDeallocateStrings(&colors);
}
. . .
```



Figure 3-6 Scroll list dialog box

Using commands, menu items, and menus in your client

The API allows you to use commands, menu items, and menus in your client's user interface. A *command* is a part of FrameMaker product or FDK client functionality that a user can invoke by typing a shortcut. A *menu item* is an instance of a command that appears on a menu. There can be several menu items for each command.

A *menu* is a list of menu items or other menus. A *menu bar* is a list of menus that appears at the top of the FrameMaker window on Windows platforms.

To use commands and menus in your client's user interface, follow these general steps:

1. Get the IDs of the FrameMaker product menu bars and menus that you want to add your client's menus and commands to.
2. Define your client's commands and add them to menus.
3. Define your client's menus and add them to FrameMaker product menus or menu bars.
4. Write an `F_ApiCommand()` callback to respond to the user invoking your client's commands.

These steps are discussed in greater detail in the following sections.

Getting the IDs of FrameMaker product menus and menu bars

To get the IDs of commands, menus, or menu bars, use `F_ApiGetNamedObject()`. The syntax for `F_ApiGetNamedObject()` is:

```
F_ObjHandleT F_ApiGetNamedObject(F_ObjHandleT parentId,  
    IntT objType,  
    StringT objName);
```

This argument	Means
<code>parentId</code>	The ID of the document, book, or session containing the object for which you want to get an ID. For commands and menus, it is always <code>FV_SessionId</code> .
<code>objType</code>	The object type. To get the ID of a command, specify <code>FO_Command</code> . To get the ID of a menu or menu bar, specify <code>FO_Menu</code> .
<code>objName</code>	The name of the command, menu, or menu bar. This name may not be the same as the label or title that appears on a menu.

The menu and command names you can specify for `objName` depend on how the user has customized the menus. The [Files] section of the `maker.ini` file

specifies the location of the menu and command configuration files that list FrameMaker's menus and commands.

The following table lists some FrameMaker product menus and the names you use to specify them:

Menu title	Name
Edit	EditMenu
Element	ElementMenu
File	FileMenu
Format	FormatMenu
Graphics	GraphicsMenu
Special	SpecialMenu
Table	TableMenu
View	ViewMenu
Help	!HelpMenu

The following table lists the names of some FrameMaker product menu bars. Menu bar names starting with an exclamation point (!) can't be removed by the user.

FrameMaker product menu bar	Name
Menu bar for documents (complete menus)	!MakerMainMenu
Menu bar for documents (quick menus)	!QuickMakerMainMenu
Menu bar for books (complete menus)	!BookMainMenu
Menu bar for books (quick menus)	!QuickBookMainMenu
View-only menu bar	!ViewOnlyMainMenu

Example

The following code gets the ID of the Edit menu and the view-only menu bar:

```
...  
F_ObjHandleT editMenuId, viewOnlyMenuBarId;  
editMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,  
                                "EditMenu");  
viewOnlyMenuBarId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,  
                                         "!ViewOnlyMainMenu");  
...
```

Defining commands and adding them to menus

To define a command and add it to a menu, use `F_ApiDefineAndAddCommand()`.

The syntax for `F_ApiDefineAndAddCommand()` is:

```
F_ObjHandleT F_ApiDefineAndAddCommand(IntT cmd,  
                                       F_ObjHandleT toMenuId,  
                                       StringT name,  
                                       StringT label,  
                                       StringT shortcut);
```

This argument	Means
<code>cmd</code>	The integer that the FrameMaker product passes to your client's <code>F_ApiCommand()</code> function when the user chooses the menu item or types the keyboard shortcut for the command.
<code>toMenuId</code>	The ID of the menu to which to add the command.
<code>name</code>	A unique name to identify the command.
<code>label</code>	The title of the command as it appears on the menu.
<code>shortcut</code>	The keyboard shortcut sequence. Many FrameMaker product commands use shortcuts beginning with Esc (!). To specify Esc when you create a command, use <code>\\!</code> in the string you pass to <code>shortcut</code> .

`F_ApiDefineAndAddCommand()` returns the ID of the command it creates.

Example

The following code defines a command with the shortcut Esc N L and adds it to the Utilities menu:

```

. . .
#define NUMBER_LINES 1
F_ObjHandleT utilsMenuId, cmdId;

utilsMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
                                  "UtilitiesMenu");
cmdId = F_ApiDefineAndAddCommand(NUMBER_LINES, utilsMenuId,
                                  "NumberLines", "Number lines", "\\!NL");
. . .

```

Figure 3-7 *Utilities menu with client-defined menu item*

Defining and adding menus

To define a menu and add it to a menu bar or another menu, use `F_ApiDefineAndAddMenu()`. The syntax for `F_ApiDefineAndAddMenu()` is:

```

F_ObjHandleT F_ApiDefineAndAddMenu(F_ObjHandleT toMenuId,
    StringT name,
    StringT label);

```

This argument	Means
<code>toMenuId</code>	The ID of the menu or menu bar to which to add the new menu
<code>name</code>	A unique name that identifies the new menu
<code>label</code>	The title of the new menu as it appears on the menu or menu bar

`F_ApiDefineAndAddMenu()` returns the ID of the menu it creates.

If you specify a menu bar ID for `toMenuId`, the FrameMaker product implements the new menu as a pull-down menu. If you specify a pull-down or a pop-up menu ID for `toMenuId`, the FrameMaker product implements the new menu as a pull-right menu.

.....
IMPORTANT: *Your menu appears only on the menu bar you specify. For example, if you add a menu only to the !MakerMainMenu menu bar, the menu will not appear if the user switches to quick menus. For your menu to appear after the user has switched to quick menus, you must also add it to the !QuickMakerMainMenu menu bar.*
.....

Adding commands to a menu that you have created

To add a command to a menu that you have created, call `F_ApiDefineAndAddCommand()` with `toMenuId` set to the ID returned by the `F_ApiDefineAndAddMenu()` call that created the menu. For example, the following code defines a menu and adds it to the FrameMaker document menu bar. Then it adds some commands to the menu.

```
. . .
#define CHECK    1
#define PRINT    2
F_ObjHandleT menubarId, menuId, cmd1Id, cmd2Id;

/* Get the ID of the FrameMaker main menu bar. */
menubarId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
                               "!MakerMainMenu");

/* Define and add the menu to the main menu. */
menuId = F_ApiDefineAndAddMenu(menubarId, "GrammarMenu",
                               "Grammar");

/* Define some commands and add them to the menu. */
cmd1Id = F_ApiDefineAndAddCommand(CHECK, menuId,
                                  "CheckGrammar", "Check Grammar", "\\!CG");
cmd2Id = F_ApiDefineAndAddCommand(PRINT, menuId,
                                  "PrintErrors", "Print Errors", "\\!PE");
. . .
```

Figure 3-8 *FrameMaker main menu bar and a client-defined menu*

Example

The following code defines a menu and adds it to the Edit menu:

```

. . .
#define CHECK    1
#define PRINT    2
F_ObjHandleT editMenuId, menuId, cmd1Id, cmd2Id;

/* Get the ID of the edit menu. */
editMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
                                "EditMenu");

/* Define the menu and add it to the Edit menu. */
menuId = F_ApiDefineAndAddMenu(editMenuId, "GrammarMenu",
                               "Grammar");

/* Define some commands and add them to the menu. */
cmd1Id = F_ApiDefineAndAddCommand(CHECK, menuId,
                                  "CheckGrammar", "Check Grammar", "\\!CG");
cmd2Id = F_ApiDefineAndAddCommand(PRINT, menuId,
                                  "PrintErrors", "Print Errors", "\\!PE");
. . .

```

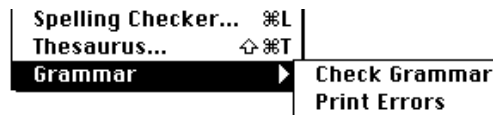


Figure 3-9 Edit menu with a client-defined pull-right menu

Responding to the user choosing a command

Whenever the user chooses a menu item or types a keyboard shortcut for a command created by your client, the FrameMaker product attempts to call your client's `F_ApiCommand()` function. Your client should define this function as follows:

```
VoidT F_ApiCommand(command)
    IntT command;
{
    /* Code to respond to command choices goes here. */
}
```

This argument	Means
<code>command</code>	The value of the <code>cmd</code> parameter in the <code>F_ApiDefineAndAddCommand()</code> call that created the command that the user chose

Example

The following client defines some commands and adds them to the Special menu. It provides an `F_ApiCommand()` function to respond to the user choosing the commands.

```
#include "fapi.h"
#define LOAD 1
#define QUERY 2

VoidT F_ApiInitialize(initialization)
    IntT initialization;
{
    F_ObjHandleT specialMenuId;

    /* Get the ID of the special menu. */
    specialMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
                                        "SpecialMenu");

    /* Define the commands and add them to the Special menu. */
    F_ApiDefineAndAddCommand(LOAD, specialMenuId,
                            "LoadDatabase", "Load Database", "");
    F_ApiDefineAndAddCommand(QUERY, specialMenuId,
                            "QueryDatabase", "Query Database", "");
}

VoidT F_ApiCommand(command)
    IntT command;
{
    switch(command)
    {
        case LOAD:          /* Code to load database goes here. */
            break;
        case QUERY:        /* Code to query database goes here. */
            break;
    }
}
```

Replacing FrameMaker product menus and commands

You can replace FrameMaker product menus and commands with your own menus and commands by calling `F_ApiDefineAndAddCommand()` and `F_ApiDefineAndAddMenu()` with the `name` parameter set to the name of a FrameMaker product menu or command.

For example, the following code replaces the FrameMaker product `Print` command:

```
. . .
#define PRINT_CMD 223
F_ObjHandleT fileMenuId, printCmdId;
fileMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
                                "FileMenu");
printCmdId = F_ApiDefineAndAddCommand(PRINT_CMD, fileMenuId,
                                      "Print", "Print...", "\\!fp");
. . .
```

If you use this code, the `Print` command appears on the `File` menu exactly as it normally would. However, when the user chooses it, the FrameMaker product never executes the `Print` operation. Instead, it calls your client's `F_ApiCommand()` callback with `command` set to `PRINT_CMD`. The `F_ApiCommand()` callback can execute your own version of the `Print` operation. For example, it can set the default number of copies to `1` and then call `F_ApiSilentPrintDoc()` to print the document. This prevents the user from printing more than one copy of a document at a time.

Allowing users to configure your client's interface

When you call `F_ApiDefineAndAddCommand()` and specify the name of a command that is already defined in the user's menu configuration files, the FrameMaker product gives precedence to the definition in the configuration files. If the configuration files assign a label or a shortcut to the command, the FrameMaker product uses it instead of the one you specify. If the command is already a menu item, the FrameMaker product ignores the menu that you specify and leaves the menu item where it is.

For example, if the Print command is already defined and appears on the File menu, the following code has the same effect as the sample code in the previous section:

```
. . .
#define PRINT_CMD 223
F_ObjHandleT printCmdId, bogusMenuId = 12345;
printCmdId = F_ApiDefineAndAddCommand(PRINT_CMD, bogusMenuId,
                                     "Print", "This is ignored", "This too");
. . .
```

If you use this code, the Print command appears on the File menu exactly as it normally does.

Because FrameMaker products give precedence to the labels, shortcuts, and menu item locations specified by the menu configuration files, users can configure your client's interface. If users know the names of your client's commands, they can assign labels and shortcuts to the commands and specify where the commands appear on the menus by editing their menu configuration files.

For example, if your client defines a command with the following code:

```
. . .
F_ObjHandleT editMenuId;
editMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
                                 "EditMenu");
F_ApiDefineAndAddCommand(1, editMenuId,
                         "NumberLines", "Number Lines", "\\!NL");
. . .
```

users can make the command appear on the Special menu instead of the Edit menu by adding the following line to their menu configuration files:

```
<Command NumberLines>
<Add NumberLines <Menu SpecialMenu>>
```

If users add this line to their menu configuration files and your client does not define the NumberLines command or is not running, the NumberLines menu item appears dimmed.

.....
IMPORTANT: *Adobe Systems recommends that you document the names of your client's menus and commands so that users can customize them.*

For more information on using menu configuration files, see the online manual, *Customizing FrameMaker Products*. For more information on changing commands, menu items, and menus, see Chapter 9, “Manipulating Commands and Menus with the API.”

Using hypertext commands in your client's user interface

You can embed hypertext commands in markers within FrameMaker product documents. A FrameMaker product's basic set of hypertext commands allows you to establish links within and between documents and to jump from link to link.

You can lock a FrameMaker product document that contains hypertext commands so that it behaves like a command palette. For information on locking documents, see your FrameMaker product user documentation. Documents have a set of properties that specify their characteristics when they are locked. By setting these properties, you can change how a locked document window appears. For example, you can hide the scroll bars and the window control buttons. For a list of locked document properties, see “Document View Only properties” in the FDK Programmer's Reference guide.

FrameMaker products provide a special hypertext command, `message apiclient`, that can send messages to your client. With this command, you can create an extremely flexible user interface. Your client only needs to define responses for the hypertext messages that are sent to it. Users and hypertext document designers can set up the interface that sends the messages. The `message apiclient` hypertext command is especially useful for setting up command palettes for your client.

To use the `message apiclient` hypertext command in your client's interface, follow the general steps below:

- 1 *Set up the hypertext commands.*
- 2 *Create a function named `F_ApiMessage()` in your client to respond to the user clicking a hypertext marker that contains a `message apiclient` command.*

These steps are discussed in greater detail in the following sections.

Setting up hypertext commands

The syntax for message *apiclient* is:

```
message apiclient yourmessage
```

This argument	Means
<i>apiclient</i>	The name under which the client is registered with the FrameMaker product. It is the <i>ClientName</i> specified in the [APIClients] section of the <i>maker.ini</i> file.
<i>yourmessage</i>	The string that the FrameMaker product passes to the API client.

When the user clicks a hypertext command, the FrameMaker product calls the `F_ApiMessage()` function of the client specified by *apiclient* and passes the string specified by *yourmessage* to the client.

Responding to message *apiclient* commands

To respond to the message *apiclient* hypertext command, your client must define `F_ApiMessage()` as follows:

```
VoidT F_ApiMessage(message, docId, objId)
    StringT message;
    F_ObjHandleT docId;
    F_ObjHandleT objId;
{
/* Code to respond to hypertext message goes here. */
}
```

This argument	Means
<i>message</i>	The string from the hypertext command message
<i>docId</i>	The ID of the document containing the hypertext marker
<i>objId</i>	The ID of the hypertext marker the user clicked

Example

Suppose you want to create a command palette with two arrows in it. When the user clicks an arrow, it changes the fill pattern of a selected graphic object in the active document. To make this command palette, create a document with the graphics shown in Figure 3-10.

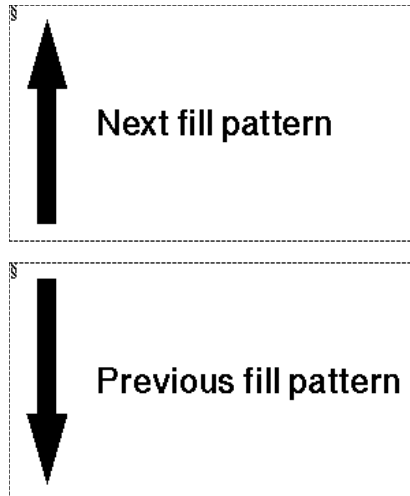


Figure 3-10 *Example hypertext markers*

Assuming your client is registered with the FrameMaker product as `myapi`, insert the following hypertext markers into the document:

- In the text column around the up arrow: `message myapi 1`
- In the text column around the down arrow: `message myapi 2`

Save the document in View Only format.

To respond to the user clicking one of the arrows, add the following code to your client:

```

. . .
#define UPARROW 1
#define DOWNARROW 2

VoidT F_ApiMessage(message, docId, objId)
    StringT message;
    F_ObjHandleT docId;
    F_ObjHandleT objId;
{
    F_ObjHandleT targetDocId, targetGraphicId;
    IntT fillpatt;

    /* Get ID of active document. Note that View Only documents
     * are not considered active.
     */
    targetDocId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);

    /* Get ID of selected object in active document. */
    targetGraphicId = F_ApiGetId(targetDocId, targetDocId,
        FP_FirstSelectedGraphicInDoc);
    if(!targetGraphicId) return;

    /* Get selected object's current fill pattern. */
    fillpatt = F_ApiGetInt(targetDocId, targetGraphicId, FP_Fill);

    switch(atoi(message))
    {
        case UPARROW:
            /* Only 16 patterns available, so reset at 16. */
            if (++fillpatt == 16) fillpatt = 0; break;
        case DOWNARROW:
            if (--fillpatt == 0) fillpatt = 15; break;
    }

    /* Set the fill pattern of the selected graphic. */
    F_ApiSetInt(targetDocId, targetGraphicId, FP_Fill, fillpatt);
}

. . .

```

For this example client to work, you should have the View Only document and one normal document open. Select a graphic in the normal document, then click one of the hypertext markers in the View Only document.

Responding to user-initiated events or FrameMaker product operations

Your client can respond when the user initiates certain events or FrameMaker product operations. For example, you may want your client to archive an extra copy of a document each time the user saves it, or to display a special dialog box when the user exits a document.

To make your client respond to events, follow these general steps:

- 1 *Request notification for the events.*
- 2 *Create a callback function named `F_ApiNotify()` to respond to the events.*

These steps are discussed in greater detail in the following sections.

Requesting notification for events

To receive notification for a particular event, your client must request it by calling `F_ApiNotification()`.

The syntax for `F_ApiNotification()` is:

```
IntT F_ApiNotification(IntT notification,  
                      IntT state);
```

This argument	Means
<code>notification</code>	A constant that specifies the notification point. See the table below for a list of the constants.
<code>state</code>	Specifies whether to turn notification for the notification point on or off. Specify <code>True</code> to request notification or <code>False</code> to turn notification off.

For most events, there are several *notification points*, or stages in the event when the FrameMaker product can notify your client. For example, when the FrameMaker product saves a file, there are two notification points: one immediately before and one immediately after it saves the file. The following table lists the notification points and constants the FrameMaker product passes to `F_ApiNotify()` for some events.

Event or operation	Notification points	Notification constants
Frame binary document opened	Before checking the type of the file to be opened	FA_Note_PreFileType
	After checking the type of the file to be opened	FA_Note_PostFileType
	Before opening the file	FA_Note_PreOpenDoc
	After opening the file	FA_Note_PostOpenDoc
Filterable document opened	Before checking the type of the file to be opened	FA_Note_FilterIn
Document saved in Frame binary format	Before saving the document	FA_Note_PreSaveDoc
	After saving the document	FA_Note_PostSaveDoc
Document saved as filterable type	Before saving the document	FA_Note_FilterOut
Document exited	Before exiting the document	FA_Note_PreQuitDoc
	After exiting the document	FA_Note_PostQuitDoc

For a complete list of events and notification points, see “`F_ApiNotification()`” in the FDK Programmer’s Reference guide.

You can request notification in your client’s `F_ApiInitialize()` callback or anywhere you want in your client.

Example

Suppose you want a FrameMaker product to notify your client whenever the user exits a document. To request this notification when your client is first started, write the `F_ApiInitialize()` callback as follows:

```
. . .  
VoidT F_ApiInitialize(initialization)  
    IntT initialization;  
{  
    /* Request notification for exit. */  
    if (initialization == FA_Init_First)  
        F_ApiNotification(FA_Note_PreQuitDoc, True);  
}  
. . .
```

Requesting notification for API filters

API client filters do not need to request notification. To receive filter notifications, filters only need to be registered with the FrameMaker product. If they are correctly registered, filters receive the following notifications:

This type of filter	Receives this notification
Import	FA_Note_FilterIn
Export	FA_Note_FilterOut

For more information on writing client filters, see “Writing filter clients” on page 476 of the *FDK Programmer's Guide*. For more information on registering filters, see the *FDK Platform Guide* for your platform.

Adding the `F_ApiNotify()` callback

The FrameMaker product notifies your client of events for which it has requested notification by calling its `F_ApiNotify()` function. Your client should define `F_ApiNotify()` as follows:

```
VoidT F_ApiNotify(notification, docId, sparm, iparm)
    IntT notification;
    F_ObjHandleT docId;
    StringT sparm;
    IntT iparm;
{
    /* Code that responds to notifications goes here. */
}
```

This argument	Means
<code>notification</code>	A constant that indicates the event and the notification point (see the table on page 220 for a list of the constants).
<code>docId</code>	The ID of the active document when the event occurs. For filters, the document into which the filter should import its data; if this is zero, the filter must create a new document.
<code>sparm</code>	The string, if any, associated with the notification. For example, if the notification is for an Open or Save, <code>sparm</code> specifies the pathname of the affected file. If the notification is for text entry, <code>sparm</code> specifies the text the user typed. Depending on how fast the user is typing, <code>sparm</code> may specify one or several characters at a time.
<code>iparm</code>	The integer associated with the notification. For example, if <code>notification</code> is <code>FA_NotePreFunction</code> or <code>FA_NotePostFunction</code> , <code>iparm</code> specifies the f-code for the command.

`F_ApiNotify()` can call API functions to get or set object properties or to initiate FrameMaker product operations. The FrameMaker product calls `F_ApiNotify()` only at the notification points for which your client has requested notification.

For example, the following code prints the name of each document the user opens to the console:

```
. . .  
VoidT F_ApiInitialize(initialization)  
    IntT initialization;  
{  
    if (initialization == FA_InitFirst)  
        F_ApiNotification(FA_Note_PostOpenDoc, True);  
}  
  
VoidT F_ApiNotify(notification, docId, sparm, iparm)  
    IntT notification;  
    F_ObjHandleT docId;  
    StringT sparm;  
    IntT iparm;  
{  
    if (notification == FA_Note_PostOpenDoc)  
        F_Printf(NULL, "The user opened: %s\n", sparm);  
}  
. . .
```

Canceling commands

Your client can cancel any command or action for which it receives a `FA_Note_PreNotificationPoint` notification. For example, if it receives the `FA_Note_PreQuitDoc` notification, it can cancel the Close command and prevent the user from closing a document.

To abort a command, call `F_ApiReturnValue()`, with the parameter set to `FR_CancelOperation`, when your client receives notification for the command. For example, the following code cancels the Exit command, preventing the user from closing any documents:

```

. . .
F_ApiNotification(FA_Note_PreQuitDoc, True);
. . .
VoidT F_ApiNotify(notification, docId, sparm, iparm)
    IntT notification;
    F_ObjHandleT docId;
    StringT sparm;
    IntT iparm;
{
    /* If user is trying to close document, cancel command. */
    if (notification == FA_Note_PreQuitDoc)
        F_ApiReturnValue(FR_CancelOperation);
}
. . .

```

Responding to text entry and actions that have no specific notifications

The API doesn't provide specific notifications for most user actions. Instead, it provides the following general notifications, which it issues for nearly every user action.

Event or operation	Notification points	Notification constants
Any user action that the FrameMaker product processes	After the FrameMaker product finishes processing the action	<code>FA_Note_BackToUser</code>
FrameMaker product command invoked or text entered in a document	Before the FrameMaker product executes the command or adds text to the document	<code>FA_Note_PreFunction</code>
	After the FrameMaker product executes the command or adds text to the document	<code>FA_Note_PostFunction</code>

The API issues the `FA_NoteBackToUser` notification after any user action the FrameMaker product processes, including insertion point changes, selection changes, and text entry. This notification is useful if you need to update a modeless dialog box containing settings that are dependent on the insertion point.

When the API issues the `FA_NoteBackToUser` notification, it indicates only that an action occurred; it does not specify which action. If you want to respond to specific actions, use the `FA_Note_PreFunction` or the `FA_Note_PostFunction` notification instead of `FA_NoteBackToUser`.

.....
IMPORTANT: When the FrameMaker product performs a book-wide command (a command that process all documents in a book), it posts an `FA_NotePreFunction` and `FA_NotePostFinction` notification for the book file, and for each document in the book. When trapping book-wide frunctions, you should check the value of `docId` to determine whether it indicates a document or the active book.

For example, if you search a book with two documents in it, the FrameMaker product posts the following funtion notifications:

- `FA_Note_PreFunction` (start searching book)
 - `FA_Note_PreFunction` (start searching first document)
 - `FA_Note_PostFunction` (stop searching first document)
 - `FA_Note_PreFunction` (start searching second document)
 - `FA_Note_PostFunction` (stop searching second document)
 - `FA_Note_PostFunction` (stop searching book)
-

When the API issues an `FA_Note_PreFunction` or `FA_Note_PostFunction` notification, it specifies the user action by setting `iparm` to a function code (f-code). An *f-code* is a hexadecimal code that specifies a command or other user action. The following table shows some common f-codes and the commands or user actions they specify.

F-code	Command or user action
<code>PGF_APPLY_TAG</code>	The user applied a paragraph format
<code>CHAR_APPLY_TAG</code>	The user applied a character format
<code>TXT_10</code>	The user set the text size to 10 points
<code>KBD_OPEN</code>	The user chose Open
<code>KBD_INPUT</code>	The user typed some text
<code>KBD_ALIGN</code>	The user chose Align

For a complete list of f-codes, see the `fcodes.h` file shipped with the FDK.

If a user action is associated with a text string, the API passes the string to the `sparm` parameter of your client's `F_ApiNotify()` function. For example, when the user types text, the API sets `sparm` to the text the user typed.

The following table lists some f-codes and the strings that are associated with them.

F-code	Associated string that the API passes to sparm
PGF_APPLY_TAG	The name of the paragraph format the user applied.
CHAR_APPLY_TAG	The name of the character format the user applied.
KBD_INPUT	The text the user typed. It can be one or more characters depending on how fast the user types.
TXT_FAMILY_AND_VARIATION	The name of the font family the user chose.

Your client can cancel any action for which it receives the `FA_Note_PreFunction` notification by calling `F_ApiReturnValue()` with `retVal` set to `FR_CancelOperation`. Your client can even cancel text entry.

For example, the following code intercepts any text the user attempts to type in a document and prints it to the console:

```

. . .
#include "fcodes.h"
/* Turn on notification. */
F_ApiNotification(FA_Note_PreFunction, True);
. . .
VoidT F_ApiNotify(notification, docId, sparm, iparm)
    IntT notification;
    F_ObjHandleT docId;
    StringT sparm;
    IntT iparm;
{
    if (notification == FA_Note_PreFunction
        && iparm == KBD_INPUT)
    {
        F_Printf(NULL, "The user typed: %s\n", sparm);
        /* Prevent text from being added to document. */
        F_ApiReturnValue(FR_CancelOperation);
    }
}
. . .

```

Responding to events initiated by API clients

A FrameMaker product notifies your client of any event that it has requested notification for. The event can be initiated directly by the user or by another client.

The Frame API provides a set of functions that allow API clients to programmatically execute Open, Save, and several other FrameMaker product operations. For more information on these functions, see Chapter 4, “Executing Commands with API Functions.” When a client executes an operation with one of these functions, the FrameMaker product notifies all the other API clients that have requested notification for that event¹. It does not, however, notify the client that executed the operation. For example, to have your client automatically make an additional copy of a document when the user saves it, use the following code:

```

. . .
/* Turn on notification. */
F_ApiNotification(FA_Note_PostSaveDoc, True);
. . .
VoidT F_ApiNotify(notification, docId, sparm, iparm)
    IntT notification;
    F_ObjHandleT docId;
    StringT sparm;
    IntT iparm;
{
    /* After the document has been saved, save another copy. */
    if (notification == FA_Note_PostSaveDoc)
        F_ApiSimpleSave(docId, "mybackup.doc", False);
}
. . .

```

In the example above, `F_ApiNotify()`, which responds to a Save notification, uses `F_ApiSimpleSave()` to execute a Save operation itself. This does not result in infinite recursion because the FrameMaker product does not notify the client of the Save operation that it executes itself.

.....

1. An API client can explicitly instruct a FrameMaker product to suppress notifications to other API clients when it opens or saves a file by setting the `FS_DontNotifyAPIClients` property of the Open or Save script to `True`. For more information on properties in the Open and Save scripts, see “`F_ApiGetOpenDefaultParams()`” and “`F_ApiGetSaveDefaultParams()`” in the FDK Programmer’s Reference guide.

Handling notification for Open operations

The Open operation is more complex than most other operations. A FrameMaker product does the following when it opens a file:

- 1 *Determines whether the file is filterable.*
If the file is filterable, the FrameMaker product issues the `FA_Note_FilterIn` notification to the appropriate filter and abandons the Open operation. It is up to the filter to finish opening the file. No other client receives any notification.

If the file is not filterable, the FrameMaker product continues with the Open operation.
- 2 *Issues an `FA_Note_PreFileType` notification to all clients that have requested it.*
This allows clients to uncompress a file if it is compressed, check it out if it is under version control, or perform other operations that may change its type.
- 3 *Checks the file's type.*
If the file is a type that the FrameMaker product can't open, the FrameMaker product displays a warning and cancels the Open operation. If the file is from a previous version of a FrameMaker product, it prompts the user to convert the file or cancel the Open operation.
- 4 *Issues an `FA_Note_PostFileType` notification to all clients that have requested it.*
- 5 *Determines whether the file is a document or a book, and whether its format is Frame binary or MIF.*
- 6 *Issues an `FA_Note_PreOpenDoc`, `FA_Note_PreOpenBook`, `FA_Note_PreOpenMIF`, or `FA_Note_PreOpenBookMIF` notification.*
- 7 *Opens the document or book.*
If the document or book is MIF, the FrameMaker product converts it.
- 8 *Issues an `FA_Note_PostOpenDoc`, `FA_Note_PostOpenMIF`, `FA_Note_PostOpenBook`, or `FA_Note_PostOpenBookMIF` notification.*
Normally, you don't request the `FA_Note_PreFileType` and `FA_Note_PostFileType` notifications, because you don't want to do anything with a file before the FrameMaker product has checked its type. However, if you want to change a file's contents after the user has selected it but before the FrameMaker product has checked its type, you should request notification for the `FA_Note_PreFileType` notification point.

For example, suppose you want to uncompress a compressed document file when the user opens it. Normally, when a user attempts to open a compressed file, the FrameMaker product displays an “Unrecognized type” alert and cancels the Open operation when it checks the file’s type. You must uncompress the file after the user has chosen it, but before the FrameMaker product checks its type. To do this, you could use the following code:

```

. . .
F_ApiNotification(FA_Note_PreFileType, True);
. . .
VoidT F_ApiNotify(notification, docId, sparm, iparm)
    IntT notification;
    F_ObjHandleT docId;
    StringT sparm;
    IntT iparm
{
    if (notification == FA_Note_PreFileType)
    {
        /* Code to test if file is compressed goes here. */
        F_ApiAlert("Uncompressing file.", FF_ALERT_CONTINUE_NOTE);
        /* Code to uncompress file goes here. */
    }
}
. . .

```

Implementing quick keys

FrameMaker products provide a *quick-key* interface, which allows the user to choose commands in the document Tag area. In FrameMaker, for example, the user can apply a character format by pressing Esc q c. FrameMaker displays an `f:` prompt in the Tag area. The user can then choose a character format by typing the first few letters of the format’s name and pressing Return when the format appears in the Tag area.

Your client can implement its own quick-key interface by calling `F_ApiQuickSelect()`. The syntax for `F_ApiQuickSelect()` is:

```
IntT F_ApiQuickSelect(F_ObjHandleT docId,
    StringT prompt,
    F_StringsT *stringlist);
```

This argument	Means
<code>docId</code>	The ID of the document containing the Tag area in which to display the prompt
<code>prompt</code>	The prompt that appears in the Tag area
<code>stringlist</code>	The list of strings from which the user can choose

`F_ApiQuickSelect()` returns the index of the string the user chose or `-1` if the user canceled the command.

For example, the following code implements the quick-key interface shown in Figure 3-11:

```
. . .
F_StringsT fruits;
StringT strings[3];
IntT choice;
F_ObjHandleT docId;

docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
strings[0] = (StringT)"Kumquat";
strings[1] = (StringT)"Durian";
strings[2] = (StringT)"Rambutan";
fruits.len = 3;
fruits.val = strings;
choice = F_ApiQuickSelect(docId, (StringT)"Fruit:", &fruits);

if (choice != -1)
    F_Printf(NULL, (StringT)"The user chose: %s.\n",
        strings[choice]);
. . .
```

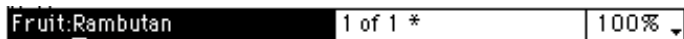


Figure 3-11 Tag area with client-defined quick key

Freeing system resources by bailing out

Instead of leaving your client idle when it's not executing, you may want to free the system resources it uses. The API provides a function named `F_ApiBailOut()`, which allows you to do this. When you call `F_ApiBailOut()`, the FrameMaker product waits until your client returns from the current callback, then exits your client, freeing the system resources it uses.

.....
IMPORTANT: *Never call `exit()`, `F_Exit()`, or `abort()` from an API client. If you call one of these functions, the function exits the FrameMaker product and unpredictable results, including data loss, may occur.*

After it exits your client, the FrameMaker product continues processing events that affect it. Your client's menus remain on the menu bar and are still valid. If your client has requested notification for particular events, the FrameMaker product continues to monitor those events. The FrameMaker product also monitors `message apiclient` hypertext commands that specify your client.

If the user chooses one of your client's menu items or hypertext commands, or initiates an event for which your client requested notification, the FrameMaker product restarts your client, calling its `F_ApiInitialize()` function with `initialization` set to `FA_Init_Subsequent`. After `F_ApiInitialize()` has returned control, the FrameMaker product also calls the appropriate callback function for the menu choice, hypertext command, or event.

.....
IMPORTANT: *If your client bails out, it loses all its global variable settings.*

The following code requests notification for the `FA_Note_PreQuitDoc` notification point and then bails out after the user starts the FrameMaker product. If the user subsequently closes a document, it displays dialog boxes indicating when the FrameMaker product reinitializes the client and when it issues the `FA_Note_PreQuitDoc` notification.

```
. . .  
VoidT F_ApiInitialize(initialization)  
IntT initialization;  
{  
switch (initialization){  
case FA_Init_First:  
  
    /* Request notification. */  
    F_ApiNotification(FA_Note_PreQuitDoc, True);  
  
    /* Bail out and wait for the event. */  
    F_ApiBailOut();  
    break;  
  
case FA_Init_Subsequent:  
    F_ApiAlert((StringT)"Client is reinitializing",  
              FF_ALERT_CONTINUE_NOTE);  
    break;  
    }  
}  
  
VoidT F_ApiNotify(notification, docId, filename)  
IntT notification;  
F_ObjHandleT docId;  
StringT filename;  
{  
    if (notification == FA_Note_PreQuitDoc)  
        F_ApiAlert((StringT)"Client has reinitialized.",  
                  FF_ALERT_CONTINUE_NOTE);  
}
```


Executing Commands with API Functions

.....

.....

This chapter discusses how to use Frame API functions to execute FrameMaker product commands programmatically.

The API doesn't provide a function to directly execute each of the commands available in the FrameMaker product user interface. This is because you can achieve the effect of some commands by setting object properties. For example, to set a graphic's fill pattern, you set the object's `FP_Fill` property. For more information on setting object properties, see Chapter 5, "Getting and Setting Properties."

Handling errors

When an API function fails, it stores an error code in the global variable, `FA_errno`. `FA_errno` retains the error code until another function fails and sets it or until your code explicitly sets it. To determine whether a set of API function calls has failed, initialize `FA_errno` to `FE_Success` once before all the calls and check it once after all the calls.

To find the error codes a function can generate, look up the function in the chapter, "FDK Function Reference," in the FDK Programmer's Reference. For a list of all API error codes and their meanings, see the `fapidefs.h` header file included with FDK or chapter, "Error Codes," in the FDK Programmer's Reference. In the interest of brevity, the examples in this chapter do not include error-handling code. However, you should check `FA_errno` after calling functions that set it.

Handling messages and warnings

In the user interface, some commands such as Open and Save sometimes need to prompt the user with warnings or messages. The API provides two types of functions to execute these commands:

- *Simple functions* allow you to either suppress the messages and warnings entirely or to prompt the user with them.
- *Scriptable functions* allow you to specify a response for each possible message or warning.

Using simple functions

Simple functions enable you to execute commands such as Save and Open without specifying numerous parameters. They execute these commands in either an interactive or a noninteractive mode. If you call a simple function and specify the interactive mode, the FrameMaker product behaves exactly as it would if the user had initiated the command. If a message or warning condition occurs, the FrameMaker product prompts the user. For example, if you call `F_ApiSimpleOpen()` in the interactive mode, the FrameMaker product displays the Open dialog box and prompts the user to choose a file to open. If the user chooses a text file, the FrameMaker product displays a Reading Text File dialog box.

If you are sure that executing a command won't do something undesirable, and you don't want the FrameMaker product to display error and warning messages, call a simple function in noninteractive mode. Be careful when you do this, because you may inadvertently destroy data. For example, suppose you attempt to save a file by calling `F_ApiSimpleSave()` in the noninteractive mode. If the file already exists, the FrameMaker product overwrites it without warning your client or the user. Noninteractive mode is useful for clients that need to carry out tasks without a user present.

Using scriptable functions

To specify a response for each possible message or warning that the FrameMaker product may issue while executing a command, use a scriptable function to execute the command. When you call a scriptable function, you pass it a *script* or property list that contains properties corresponding to possible messages or warnings. For most messages and warnings, you either specify a Yes, No, or Cancel response, or you can instruct the FrameMaker product to prompt the user for the response. Scriptable functions return detailed information on how the FrameMaker product executes a command. For example, the scriptable Open function `F_ApiOpen()` returns information, such as whether the file was filtered and whether an Autosave file was used.

The API provides a function named `F_ApiAllocatePropVals()`, which allocates a property list that you can use with scriptable functions. The API also provides functions that create default scripts for the different scriptable functions. You can use these functions to get a default script and then customize the script by changing individual properties.

Opening documents and books

The API provides two functions to open a document or book:

- `F_ApiSimpleOpen()` is an easy-to-use function for opening a document or book.
- `F_ApiOpen()` allows you to script the process of opening a document or book.

Opening a document or book with `F_ApiSimpleOpen()`

The syntax for `F_ApiSimpleOpen()` is:

```
F_ObjHandleT F_ApiSimpleOpen(StringT fileName,
    BoolT interactive);
```

This argument	Means
<code>fileName</code>	The absolute pathname of the file to open. For information on how filenames and paths on different platforms are expressed, see the <i>FDK Platform Guide</i> for your platform.
<code>interactive</code>	Specifies whether the FrameMaker product displays messages and warnings to the user.

If `F_ApiSimpleOpen()` is successful, it returns the ID of the `FO_Doc` or `FO_Book` object that represents the document or book that it opened. If a condition (such as a nonexistent file) makes it impossible to open a file, `F_ApiSimpleOpen()` aborts the operation and returns `0`.

If you set `interactive` to `True`, the FrameMaker product displays the Open dialog box. It uses the path specified by the session property `FP_OpenDir` as the default path. The FrameMaker product also displays all the other messages and warnings it would normally display if the user chose the Open command. For example, if a document contains fonts that are not available in the current session, the FrameMaker product displays a “Fonts Unavailable. Open Anyway?” dialog box. If the user clicks Cancel, `F_ApiSimpleOpen()` aborts the operation and returns `0`.

If you set `interactive` to `False`, the FrameMaker product does not display the Open dialog box or other messages and warnings. If it is necessary to modify a file to continue opening it, `F_ApiSimpleOpen()` aborts the operation without notifying the user, and returns `0`. For example, if a document contains fonts that are not available, `F_ApiSimpleOpen()` aborts the Open operation instead of converting the fonts.

Example

The following code opens a document named `/tmp/my.doc` and displays its ID:

```

. . .
#include "futils.h"
F_ObjHandleT docId;
UCharT msg[256];

docId = F_ApiSimpleOpen((StringT)"/tmp/my.doc", False);

if (!docId)
    F_ApiAlert((StringT)"Couldn't open.", FF_ALERT_CONTINUE_NOTE);
else
    {
        F_Printf(msg, (StringT)"my.doc's ID is 0x%x.", docId);
        F_ApiAlert(msg, FF_ALERT_CONTINUE_NOTE);
    }
. . .

```

Opening a document or book with `F_ApiOpen()`

To open a document or book and programmatically specify responses to warnings and messages that the FrameMaker product issues, use `F_ApiOpen()`. With `F_ApiOpen()`, you can specify aspects of the Open operation, such as whether to make a document visible and whether to use an Autosave file. You can specify all aspects of the operation, or you can specify some aspects and allow the user to decide others. For example, you can instruct the FrameMaker product to only open a MIF file but allow the user to choose the file.

To use `F_ApiOpen()`, you should first understand property lists and how to manipulate them directly. For more information on this subject, see “Representing object characteristics with properties” on page 65 and “Manipulating property lists directly” on page 295.

The syntax for `F_ApiOpen()` is:

```
F_ObjHandleT F_ApiOpen(StringT fileName,  
    F_PropValsT *openParamsp,  
    F_PropValsT **openReturnParamssp);
```

This argument	Means
<code>fileName</code>	The absolute pathname of the file to open. If you are using <code>F_ApiOpen()</code> to create a document, specify the template name.
<code>openParamsp</code>	A property list (script) that tells the FrameMaker product how to open the file and how to respond to errors and other conditions that arise. Use <code>F_ApiGetOpenDefaultParams()</code> or <code>F_ApiAllocatePropVals()</code> to create and allocate memory for this property list. To use the default list, specify <code>NULL</code> .
<code>openReturnParamssp</code>	A property list that returns the pathname and provides information on how the FrameMaker product opened the file.

.....
IMPORTANT: Always initialize the pointer to the property list that you specify for `openReturnParamssp` to `NULL` before you call `F_ApiOpen()`.
.....

If `F_ApiOpen()` is successful, it returns the ID of the opened document or book. Otherwise, it returns `0`.

To call `F_ApiOpen()`, do the following:

- 1 Initialize the pointer to the `openReturnParamssp` property list to `NULL`.
- 2 Create an `openParamsp` property list.
You can get a default list by calling `F_ApiGetOpenDefaultParams()`, or you can create a list from scratch.
- 3 Call `F_ApiOpen()`.
- 4 Check the Open status.
Check the returned values in the `openReturnParamssp` list for the name of the opened file and other information about how the FrameMaker product opened the file.

- 5 Deallocate memory for the `openParamsp` and `openReturnParamsp` property lists.

Use `F_ApiDeallocatePropVals()` to deallocate memory for the lists.

Steps 2, 4, and 5 are discussed in the following sections.

Creating an `openParamsp` script with `F_ApiGetOpenDefaultParams()`

If you need to specify a number of properties in the `openParamsp` property list, it is easiest to get a default list with `F_ApiGetOpenDefaultParams()` and then modify individual properties in the list.

The syntax for `F_ApiGetOpenDefaultParams()` is:

```
F_PropValsT F_ApiGetOpenDefaultParams();
```

The following table lists some of the properties in the property list returned by `F_ApiGetOpenDefaultParams()`. The first value listed for each property is the default value used in the list. You can change the list to use the other listed values. For the complete list of properties in the property list, see “`F_ApiGetOpenDefaultParams()`” in the FDK Programmer’s Reference guide.

Property	Instruction or situation and possible values
FS_ShowBrowser	Display Open dialog box.
	False: don’t display it.
	True: display it.
FS_OpenDocViewOnly	Open document as View Only.
	False: don’t open as View Only.
	True: open as View Only.
FS_NameStripe	String specifying the name that appears on the document title bar.
	NULL.
FS_NewDoc	Create a new document.
	False: open an existing document.
	True: create a new document.

For example, to get a default `openParamsp` property list and modify it so that it instructs `F_ApiOpen()` to show the Open dialog box, use the following code:

```
. . .
F_ObjHandleT docId;
F_PropValsT params, *returnParamsp = NULL;
IntT i;

/* Get a default property list. */
params = F_ApiGetOpenDefaultParams();

/* If F_ApiGetOpenDefaultParams() fails, len will be 0. */
if(params.len == 0) return;

/* Get index of FS_ShowBrowser property, then set it to True. */
i = F_ApiGetPropIndex(&params, FS_ShowBrowser);
params.val[i].propVal.u.ival = True;

/* Change default to /tmp when Open dialog box appears. */
F_ApiSetString(0, FV_SessionId, FP_OpenDir, "/tmp");

docId = F_ApiOpen("", &params, &returnParamsp);
F_ApiDeallocatePropVals(&params);
F_ApiDeallocatePropVals(returnParamsp);
. . .
```

The API allocates memory for the property list created by `F_ApiGetOpenDefaultParams()`. Use `F_ApiDeallocatePropVals()` to free the property list when you are done with it.

For another example of how to call `F_ApiOpen()` using a default property list created by `F_ApiGetOpenDefaultParams()`, see “`F_ApiGetOpenDefaultParams()`” in the FDK Programmer’s Reference guide.

Creating an `openParamsp` script from scratch

If you only need to specify a few properties when you call `F_ApiOpen()`, it is most efficient to create a property list from scratch. To create the property list, you must allocate memory for it and then set up the individual properties.

To allocate memory for the property list, use the API convenience function, `F_ApiAllocatePropVals()`. The syntax for `F_ApiAllocatePropVals()` is:

```
F_PropValsT F_ApiAllocatePropVals(IntT numProps);
```

This argument	Means
---------------	-------

<code>numProps</code>	The number of properties for which to allocate memory
-----------------------	---

For example, the following code creates an `openParamsp` property list that instructs `F_ApiOpen()` to show the Open dialog box:

```
. . .
F_ObjHandleT docId;
F_PropValsT params, *returnParamsp = NULL;

/* Allocate memory for the list. */
params = F_ApiAllocatePropVals(1);

/* Set up the FS_ShowBrowser property and set it to True. */
params.val[0].propIdent.num = FS_ShowBrowser;
params.val[0].propVal.valType = FT_Integer;
params.val[0].propVal.u.ival = True;

docId = F_ApiOpen("", &params, &returnParamsp);
F_ApiDeallocatePropVals(&params);
F_ApiDeallocatePropVals(returnParamsp);
. . .
```

The API allocates memory for the property list created by `F_ApiAllocatePropVals()`. Use `F_ApiDeallocatePropVals()` to free the property list when you are done with it.

Checking the Open status

`F_ApiOpen()` stores a pointer to a property list (`F_PropValsT` structure) in `openReturnParamspp`. To get the name of the file that `F_ApiOpen()` opened and other information about how `F_ApiOpen()` opened the file, check this property list. It includes the properties shown in the following table.

Property	Meaning and possible values
<code>FS_OpenedFileName</code>	A string that specifies the opened file's pathname. If you scripted <code>FS_ShowBrowser</code> , or the file was filtered, or you didn't specify the pathname, this pathname can be different from the one you specified in the Open script.
<code>FS_OpenNativeError</code>	The error condition. If the file is opened successfully, it is set to <code>FE_Success</code> . For a complete list of the other values it can be set to, see "F_ApiOpen()" in the FDK Programmer's Reference guide.
<code>FS_OpenStatus</code>	A bit field indicating what happened when the file was opened. For a complete list of the possible status flags, see "F_ApiOpen()" in the FDK Programmer's Reference guide.

The `FS_OpenNativeError` property and the `FA_errno` global variable indicate the result of a call to `F_ApiOpen()`. The `FS_OpenStatus` flags indicate how or why this result occurred. For example, if you attempt to open a file with `F_ApiOpen()` and the Open operation is canceled, `FS_OpenNativeError` and `FA_errno` are set to `FE_Canceled`. If the operation was canceled because the user canceled it, the `FV_UserCanceled` bit of the `FS_OpenStatus` property list is set.

The API provides a function named `F_ApiCheckStatus()`, which allows you to determine if a particular `FS_OpenStatus` bit is set. The syntax for `F_ApiCheckStatus()` is:

```
IntT F_ApiCheckStatus(F_PropValsT *p,
    IntT statusBit);
```

This argument	Means
<code>p</code>	The <code>openReturnParamspp</code> property list returned by <code>F_ApiOpen()</code>
<code>statusBit</code>	The status bit you want to test

If the specified bit is set, `F_ApiCheckStatus()` returns `True`. For example, the following code determines if an `Open` operation was canceled because a document used unavailable fonts:

```
. . .
F_ObjHandleT docId;

F_PropValsT params, *returnParamsp = NULL;

/* Get default property list. */
params = F_ApiGetOpenDefaultParams();

docId = F_ApiOpen("/tmp/my.doc", &params, &returnParamsp);
if (F_ApiCheckStatus(returnParamsp, FV_CancelFontsMapped))
    F_ApiAlert("Canceled because my.doc has unavailable fonts.",
              FF_ALERT_CONTINUE_NOTE);

/* Deallocate property lists. */
F_ApiDeallocatePropVals(&params);
F_ApiDeallocatePropVals(returnParamsp);
. . .
```

The API also provides a convenience function named `F_ApiPrintOpenStatus()`, which prints the `Open` status values to the Frame console.

`F_ApiPrintOpenStatus()` is useful for debugging clients that use `F_ApiOpen()`. For more information, see “`F_ApiPrintOpenStatus()`” in the FDK Programmer’s Reference guide.

Deallocating Open script property lists

After you are done with the `Open` script property lists, call the API convenience function, `F_ApiDeallocatePropVals()`, to free the memory they use.

The syntax for `F_ApiDeallocatePropVals()` is:

```
VoidT F_ApiDeallocatePropVals(F_PropValsT *pvp);
```

This argument	Means
<code>pvp</code>	The property list

Example

The following code opens a document named `/tmp/my.doc`. It creates a property list that instructs `F_ApiOpen()` to open the document as View Only and to display the title, `Doc`, in the title bar.

```
...
#include "fstrings.h"

F_PropValsT params, *returnParamsp = NULL;
F_ObjHandleT docId;

/* Allocate memory for Open script with two properties. */
params = F_ApiAllocatePropVals(2);
if(params.len == 0) return;

/* Force title displayed on title bar to be "Doc". */
params.val[0].propIdent.num = FS_NameStripe;
params.val[0].propVal.valType = FT_String;
params.val[0].propVal.u.sval = (StringT)F_StrCopyString("Doc");

/* Open the file as View Only. */
params.val[1].propIdent.num = FS_OpenDocViewOnly;
params.val[1].propVal.valType = FT_Integer;
params.val[1].propVal.u.ival = True;

/* Open /tmp/my.doc. */
docId = F_ApiOpen("/tmp/my.doc", &params, &returnParamsp);

/* Free memory used by the Open scripts. */
F_ApiDeallocatePropVals(&params);
F_ApiDeallocatePropVals(returnParamsp);
...
```

Creating documents

To create a new document, you can use the following functions:

- `F_ApiSimpleNewDoc()` is an easy-to-use function that allows you to specify a template and interactive or noninteractive modes.
- `F_ApiCustomDoc()` uses the FrameMaker product's default new document template and some parameters that you specify to create the new document.
- `F_ApiOpen()` allows you to script the New operation.

For information on creating books, see “*Creating a book*” on page 365. The following sections describe how to create a new document in greater detail.

Creating a document with `F_ApiSimpleNewDoc()`

To create a new document from a specific template, use `F_ApiSimpleNewDoc()`.

The syntax for `F_ApiSimpleNewDoc()` is:

```
F_ObjHandleT F_ApiSimpleNewDoc(StringT templateName,
                               IntT interactive);
```

This argument	Means
<code>templateName</code>	The absolute pathname of the template to use. For information on how filenames and paths on different platforms are expressed, see the <i>FDK Platform Guide</i> for that platform.
<code>interactive</code>	Specifies whether the FrameMaker product displays messages and warnings to the user.

If you set `interactive` to `True`, the FrameMaker product creates a document from the specified template and displays messages and warnings to the user. If you set `interactive` to `False`, the FrameMaker product does not display messages and warnings; if the FrameMaker product encounters a condition for which it normally displays a dialog box, `F_ApiSimpleNewDoc()` attempts to do what's necessary to continue creating the file.

If `F_ApiSimpleNewDoc()` is successful, it returns the ID of the document it created; otherwise, it returns `0`. You don't provide the name for the new document until you save it.

.....
IMPORTANT: If you call `F_ApiSimpleNewDoc()` with `interactive` set to `True` and the user clicks Portrait, Custom, or Landscape in the New dialog box, `F_ApiSimpleNewDoc()` does not create a document. It returns `0` and sets

FA_errno to FE_WantsPortrait, FE_WantsCustom, or FE_WantsLandscape. It is up to your client to create a portrait, custom, or landscape document by calling F_ApiCustomDoc(). For more information on creating custom documents, see “Creating a custom document,” next.

Example

The following code creates a document from the /templates/Reports/Report1 template and saves it as /tmp/mynew.doc. It then uses F_ApiSimpleSave() to save the the document. For more information on F_ApiSimpleSave(), see “Saving documents and books” on page 253

```
.....  
F_ObjHandleT docId;  
  
docId = F_ApiSimpleNewDoc("/templates/Reports/Report1", False);  
  
if (!docId)  
    F_ApiAlert("Can't create document.", FF_ALERT_CONTINUE_NOTE);  
else  
    F_ApiSimpleSave(docId, "/tmp/mynew.doc", False);  
.....
```

Creating a custom document

To create a custom new document, use F_ApiCustomDoc(). This function uses the FrameMaker product’s default new-document template to create the custom document. For more information on the default new-document template, see “Documents” on page 75.

The syntax for F_ApiCustomDoc() is:

```
F_ObjHandleT F_ApiCustomDoc(MetricT width,  
    MetricT height,  
    IntT numCols,  
    MetricT columnGap,  
    MetricT topMargin,  
    MetricT botMargin,  
    MetricT leftinsideMargin,  
    MetricT rightoutsideMargin,  
    IntT sidedness,  
    BoolT makeVisible);
```

This argument	Means
<code>width</code>	Page width. The Frame API expresses linear measurements with <code>MetricT</code> values. For more information on <code>MetricT</code> values, see chapter, “Data Types and Structures Reference,” in the FDK Programmer’s Reference guide.
<code>height</code>	Page height.
<code>numCols</code>	Default number of columns.
<code>columnGap</code>	Default column spacing.
<code>topMargin</code>	Page top margin.
<code>botMargin</code>	Page bottom margin.
<code>leftinsideMargin</code>	Left margin (for single-sided documents) or the inside margin (for double-sided documents).
<code>rightoutsideMargin</code>	Right margin (for single-sided documents) or the outside margin (for double-sided documents).
<code>sidedness</code>	Constant that specifies whether the document is single-sided or double-sided and on which side the document starts. See the following table for the list of constants.
<code>makeVisible</code>	Specifies whether to make the document visible. <code>True</code> makes it visible.

The `sidedness` argument can have any of the values shown in the following table.

sidedness constant	New document page characteristics
<code>FF_Custom_SingleSided</code>	Single-sided
<code>FF_Custom_FirstPageRight</code>	Double-sided, starting with a right page
<code>FF_Custom_FirstPageLeft</code>	Double-sided, starting with a left page

If successful, `F_ApiCustomDoc()` returns the ID of the document it created. Otherwise, it returns `0`.

Example

The following code creates a custom document with the characteristics specified in the dialog box in Figure 4-1:

```

. . .
#include "fmetrics.h"
#define in (MetricT)(65536*72) /* A Frame metric inch */

F_ObjHandleT docId;

docId = F_ApiCustomDoc(F_MetricFractMul(in,17,2), 11*in, 1,
    F_MetricFractMul(in,1,4), in, in, in, in,
    FF_Custom_SingleSided, True);
. . .
    
```

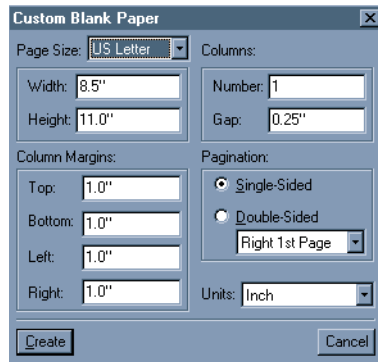


Figure 4-1 Specifications for custom document

Creating a document with F_ApiOpen()

To create a document with `F_ApiOpen()`, set the `FS_NewDoc` property in the `openParamsp` script to `True`. For the syntax of `F_ApiOpen()`, see “Opening a document or book with `F_ApiOpen()`” on page 238.

When you use `F_ApiOpen()` to create a document, set `fileName` to the name of the template that you want to use. You don’t provide the name for the new document until you save it.

For example, the following code creates a document from a template named `/tmp/template` and saves it as `/tmp/mynew.doc`.

```
. . .
F_ObjHandleT docId;
F_PropValsT params, *returnParamsp = NULL;

params = F_ApiAllocatePropVals(1);
if(params.len == 0) return;

/* Set up the FS_NewDoc property and set it to True. */
params.val[0].propIdent.num = FS_NewDoc;
params.val[0].propVal.valType = FT_Integer;
params.val[0].propVal.u.ival = True;

docId = F_ApiOpen("/tmp/template", &params, &returnParamsp);

/* See "Saving documents and books" on page 253 for syntax. */
F_ApiSimpleSave(docId, "/tmp/mynew.doc", False);

/* Deallocate property lists. */
F_ApiDeallocatePropVals(&params);
F_ApiDeallocatePropVals(returnParamsp);
. . .
```

.....

IMPORTANT: If you are creating a document with `F_ApiOpen()` and you display the New dialog box (by setting `FS_ShowBrowser` to `True`), the user may click Portrait, Custom, or Landscape. If this occurs, `F_ApiOpen()` does not create a new document. It returns 0 and sets `FA_errno` to `FE_WantsPortrait`, `FE_WantsCustom`, or `FE_WantsLandscape`. It is up to your client to create a portrait, custom, or landscape document by calling `F_ApiCustomDoc()`.

.....

Printing documents and books

To print a document or book, use `F_ApiSilentPrintDoc()`.
`F_ApiSilentPrintDoc()` uses the default print settings for a document. The default print settings are the settings that appear in the Print dialog box when the user attempts to print the document in the user interface.

The syntax for `F_ApiSilentPrintDoc()` is:

```
IntT F_ApiSilentPrintDoc(F_ObjHandleT docId);
```

This argument	Means
---------------	-------

<code>docId</code>	The ID of the document or book to print
--------------------	---

When you call `F_ApiSilentPrintDoc()`, the FrameMaker product doesn't notify the user about error or warning conditions that occur when it attempts to print. To determine whether an error occurred, check `FA_errno`.

Changing the print settings for a document

When you print a document in the user interface, you can change the print settings in the Print dialog box. FrameMaker products save most print settings with a document. For example, if you set the scale to 90 percent and print the document in the same session or save the document, the default setting for the scale will be 90 percent. Similarly, if an API client calls `F_ApiSilentPrintDoc()` to print the document, the scale will be 90 percent, if the client doesn't change it.

The API represents a document's print settings with a set of document properties. For example, a document's `FP_PrintNumCopies` property specifies the number of copies of the document to print. To change a print setting programmatically, you change the property that represents it. For more information on changing properties, see Chapter 5, "Getting and Setting Properties." For a list of document print properties, see "Document print properties" in the FDK Programmer's Reference guide.

Examples

The following code opens a document named `/tmp/my.doc` and prints it using the default print settings:

```
...  
F_ObjHandleT docId;  
docId = F_ApiSimpleOpen("/tmp/my.doc", False);  
F_ApiSilentPrintDoc(docId);  
...
```

The following code opens `/tmp/my.doc` and modifies its default print settings so that the FrameMaker product will print two copies of the document to a printer named `ps2`. It does this by setting the document properties that specify the number of copies (`FP_PrintNumCopies`) and the printer (`FP_PrinterName`) to `2` and `ps2`, respectively:

```
. . .
F_ObjHandleT docId;

/* Open the document. */
docId = F_ApiSimpleOpen("/tmp/my.doc", False);

/* Change my.doc's print properties. */
F_ApiSetInt(FV_SessionId, docId, FP_PrintNumCopies, 2);
F_ApiSetString(FV_SessionId, docId, FP_PrinterName, "ps2");

F_ApiSilentPrintDoc(docId);
. . .
```

If you save `/tmp/my.doc` or attempt to print it within the same session, the default printer will be `ps2` and the default number of copies will be `2` unless your client or the user changes the values of `FP_PrinterName` and `FP_PrintNumCopies`.

Saving documents and books

To save a document or book, use one of the following functions:

- `F_ApiSimpleSave()` is an easy-to-use function for saving a document or book.
- `F_ApiSave()` allows you to script the process for saving a document or book.

Saving a document or book with `F_ApiSimpleSave()`

The syntax for `F_ApiSimpleSave()` is:

```
F_ObjHandleT F_ApiSimpleSave(F_ObjHandleT docId,  
    StringT saveAsName,  
    IntT interactive);
```

This argument	Means
<code>docId</code>	ID of the document or book to save.
<code>saveAsName</code>	Name of the pathname to save the document or book to. For information on how filenames and paths on different platforms are represented, see the <i>FDK Platform Guide</i> for that platform.
<code>interactive</code>	Specifies whether the FrameMaker product displays messages and warnings to the user (<code>True</code> to display messages and warnings).

If you set `interactive` to `False` and you specify the document or book's current name, the FrameMaker product saves the document or book under its current name. If you specify another filename for `saveAsName`, the FrameMaker product saves the document or book to that filename.

If you set `interactive` to `True`, the FrameMaker product displays the Save dialog box and allows the user to choose a filename. The document or book's current name appears as the default name.

If `F_ApiSimpleSave()` is successful, it returns the ID of the document it saved. If you save the document under its current name, the returned ID is the same ID you specified in the `docId` parameter. If you specify another filename for `saveAsName`, the returned ID is the ID of the new document. If `F_ApiSimpleSave()` can't save the file, it returns `0`.

Example

The following code opens and then saves a document named `/tmp/my.doc`. After it has saved the document as `/tmp/my.doc`, it saves a copy of it as `mynew.doc`:

```

. . .
#include "futils.h"
F_ObjHandleT mydocId, mynewdocId;
UCharT msg[256];

mydocId = F_ApiSimpleOpen("/tmp/my.doc", False);

/* Save my.doc as itself. */
F_ApiSimpleSave(mydocId, "/tmp/my.doc", False);

/* Save my.doc as mynew.doc. */
mynewdocId = F_ApiSimpleSave(mydocId, "/tmp/mynew.doc", False);

/* If the Save As was successful, display ID of mynew.doc. */
if (!mynewdocId)
    F_ApiAlert("Couldn't save as mynew.doc.",
               FF_ALERT_CONTINUE_NOTE);
else {
    F_Sprintf(msg, "The ID of mynew.doc is 0x%x.", mynewdocId);
    F_ApiAlert(msg, FF_ALERT_CONTINUE_NOTE);
}
. . .

```

Saving a document or book with `F_ApiSave()`

To save a document or book and specify responses to warnings and messages that the FrameMaker product issues, use the scriptable save function, `F_ApiSave()`. With `F_ApiSave()`, you can specify aspects of the Save operation, such as the file format (for example, MIF or Text Only). You can specify all aspects, or you can specify some and allow the user to decide others. For example, you can specify that the FrameMaker product should save a document as Text Only, but allow the user to decide how to convert the document's tables to text.

The syntax for `F_ApiSave()` is:

```
F_ObjHandleT F_ApiSave(F_ObjHandleT docId,
    StringT saveAsName,
    F_PropValsT *saveParamsp,
    F_PropValsT **saveReturnParamssp);
```

This argument	Means
<code>docId</code>	The ID of the document or book to save.
<code>saveAsName</code>	The pathname to save the document or book to.
<code>saveParamsp</code>	A property list that tells the FrameMaker product how to save the file and how to respond to errors and other conditions. Use <code>F_ApiGetSaveDefaultParams()</code> or <code>F_ApiAllocatePropVals()</code> to create and allocate memory for this property list. To use the default list, specify <code>NULL</code> .
<code>saveReturnParamssp</code>	A property list that returns information about how the FrameMaker product saved the file.

.....
IMPORTANT: Always initialize the pointer to the property list that you specify for `saveReturnParamssp` to `NULL` before you call `F_ApiSave()`.

If `F_ApiSave()` is successful, it returns the ID of the document or book it saved. If `F_ApiSave()` performs a Save operation, it returns the ID that you specified in the `docId` parameter. If `F_ApiSave()` performs a Save As operation, it returns the ID of the new document or book. If `F_ApiSave()` can't save a file, it returns `0`.

To call `F_ApiSave()`, do the following:

- 1 *Initialize the pointer to the `saveReturnParamssp` property list to `NULL`.*
- 2 *Create a `saveParamsp` property list.*
 You can get a default list by calling `F_ApiGetSaveDefaultParams()`, or you can create a list from scratch.
- 3 *Call `F_ApiSave()`.*
- 4 *Check the Save status.*
 Check the returned values in the `saveReturnParamssp` list for the name of the saved file and other information about how the FrameMaker product saved the file.
- 5 *Deallocate the `saveParamsp` and `saveReturnParamssp` property lists.*
 Steps 2, 4, and 5 are discussed in the following sections.

Creating a saveParamsp script with F_ApiGetSaveDefaultParams()

The API provides a function named `F_ApiGetSaveDefaultParams()` that creates a default `saveParamsp` property list. If you are setting a number of properties, it is easiest to use `F_ApiGetSaveDefaultParams()` to get a default property list and then change individual properties as needed.

The syntax for `F_ApiGetSaveDefaultParams()` is:

```
F_PropValsT F_ApiGetSaveDefaultParams();
```

The following table lists some of the properties in the property list returned by `F_ApiGetSaveDefaultParams()`. The first value listed for each property is the default value returned by `F_ApiGetSaveDefaultParams()`. You can change the list to use the other listed values.

Property	Meaning and possible values
FS_FileType	Specifies the type of file to save to
	FV_SaveFmtBinary: save in Frame binary format
	FV_SaveFmtInterchange: save as MIF
	FV_SaveFmtStationery: save in Stationery format
	FV_SaveFmtViewOnly: save as View Only
	FV_SaveFmtText: save as Text Only
	FV_SaveFmtSgml: save as SGML
FS_AlertUserAboutFailure	Specifies whether to notify the user if something unusual occurs while the file is being saved
	False: don't notify user
	True: notify user
FS_SaveMode	Specifies whether to use Save or Save As mode
	FV_ModeSaveAs: use Save As mode
	FV_ModeSave: use Save mode

For the complete property list returned by `F_ApiGetSaveDefaultParams()`, see “`F_ApiGetSaveDefaultParams()`” in the FDK Programmer’s Reference guide.

For example, to get a default `saveParamsp` property list and modify it so that it instructs `F_ApiSave()` to save the active document as Text Only, use the following code:

```
. . .
F_PropValsT params, *returnParamsp = NULL;
F_ObjHandleT mydocId;
IntT i;

/* Get the ID of the active document. */
mydocId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);

/* Get default property list. */
params = F_ApiGetSaveDefaultParams();

/* Get index of FS_FileType property and set it to Text Only. */
i = F_ApiGetPropIndex(&params, FS_FileType);
params.val[i].propVal.u.ival = FV_SaveFmtText;

/* Save to text only file named my.txt. */
F_ApiSave(mydocId, "/tmp/my.txt", &params, &returnParamsp);

/* Deallocate property lists. */
F_ApiDeallocatePropVals(&params);
F_ApiDeallocatePropVals(returnParamsp);
. . .
```

The API allocates memory for the property list created by `F_ApiGetSaveDefaultParams()`. Use `F_ApiDeallocatePropVals()` to free the property list when you are done with it.

Creating a `saveParamsp` script from scratch

If you want to specify only a few properties when you call `F_ApiSave()`, it is most efficient to create a property list from scratch. To create the property list, you must allocate memory for it, and then set up the individual properties.

Use the API convenience function, `F_ApiAllocatePropVals()`, to allocate memory for the property list. For example, the following code creates a `saveParamsp` property list that instructs `F_ApiSave()` to save a file as text only:

```
. . .
F_PropValsT params, *returnParamsp = NULL;
F_ObjHandleT mydocId;

/* Get the ID of the active document. */
mydocId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);

/* Allocate memory for the list. */
params = F_ApiAllocatePropVals(1);

/* Set up the FS_FileType property and set it to True. */
params.val[0].propIdent.num = FS_FileType;
params.val[0].propVal.valType = FT_Integer;
params.val[0].propVal.u.ival = FV_SaveFmtText;

F_ApiSave(mydocId, "/tmp/my.txt", &params, &returnParamsp);
. . .
```

Checking Save status

`F_ApiSave()` stores a pointer to a property list in `saveReturnParamsp`. This property list provides information on how the FrameMaker product saved the file. It includes the properties shown in the following table.

Property	Meaning and Possible Values
<code>FS_SavedFileName</code>	A string that specifies the saved file's pathname.
<code>FS_SaveNativeError</code>	The error condition. If the file is saved successfully, it is set to <code>FE_Success</code> . For a complete list of the other values it can be set to, see “ <code>F_ApiSave()</code> ” in the FDK Programmer's Reference guide.
<code>FS_SaveStatus</code>	A bit field indicating what happened when the file was saved. For a complete list of the possible status flags, see “ <code>F_ApiSave()</code> ” in the FDK Programmer's Reference guide.

The `FS_SaveNativeError` property and the `FA_errno` value indicate the result of the call to `F_ApiSave()`. The `FS_SaveStatus` flags indicate how or why this result occurred.

To determine if a particular `FS_SaveStatus` bit is set, use `F_ApiCheckStatus()`.

Example

The following code opens `/tmp/my.doc` and saves it as a View Only document named `/tmp/viewonly.doc`. It gets the name of the saved file from the returned property list and displays it.

```

. . .
#include "futils.h"

IntT i;
UCharT msg[1024];
F_PropValsT params, *returnParamsp = NULL;
F_ObjHandleT mydocId, viewonlydocId;

params = F_ApiAllocatePropVals(1);

mydocId = F_ApiSimpleOpen("/tmp/my.doc", False);
if(!mydocId) return;

/* Set file type to View Only. */
params.val[0].propIdent.num = FS_FileType;
params.val[0].propVal.valType = FT_Integer;
params.val[0].propVal.u.ival = FV_SaveFmtViewOnly;

/* Save document as viewonly.doc. */
viewonlydocId = F_ApiSave(mydocId, "/tmp/viewonly.doc",
                        &params, &returnParamsp);

/* Get index of property specifying filename and display it. */
i = F_ApiGetPropIndex(returnParamsp, FS_SavedFileName);
F_Sprintf(msg, "Saved: %s",
          returnParamsp->val[i].propVal.u.sval);
F_ApiAlert(msg, FF_ALERT_CONTINUE_NOTE);

/* Deallocate Save scripts. */
F_ApiDeallocatePropVals(&params);
F_ApiDeallocatePropVals(returnParamsp);
. . .

```

Closing documents and books

To close a document or book, use `F_ApiClose()`.

The syntax for `F_ApiClose()` is:

```
IntT F_ApiClose(F_ObjHandleT Id,
               IntT flags);
```

This argument	Means
Id	The ID of the document, book, or session to close. To close the session, specify <code>FV_SessionId</code> .
flags	Specifies whether to abort or to close open documents or books if they have unsaved changes. Set the <code>FF_CLOSE_MODIFIED</code> flag to close open documents and books regardless of their state.

`F_ApiClose()` behaves somewhat differently than the `Close` command in the user interface. If there are unsaved changes in a file and you set `FF_CLOSE_MODIFIED` for the `flags` argument, `F_ApiClose()` abandons the changes and closes the file anyway. If you set `flags` to 0, `F_ApiClose()` aborts the `Close` operation and returns `FE_DocModified`.

.....
IMPORTANT: If you are closing an individual document, make sure `Id` specifies a valid document ID and not 0. If `Id` is set to 0, `F_ApiClose()` quits the Frame session (because `FV_SessionId` is defined as 0).

Examples

The following code closes the active document. If the document has unsaved changes, the client prompts the user.

```
. . .
F_ObjHandleT docId;
IntT resp = 0;

/* Get the ID of active document. Return if there isn't one. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
if(!docId) return;

/* See if document has been modified. */
if (F_ApiGetInt(FV_SessionId, docId, FP_DocIsModified))
    resp = F_ApiAlert("Document was changed, close it anyway?",
                      FF_ALERT_OK_DEFAULT);

if (!resp) F_ApiClose(docId, FF_CLOSE_MODIFIED);
. . .
```

The following code closes the active document unless it has unsaved changes:

```
. . .
F_ObjHandleT docId;

docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
if(!docId) return;
if (F_ApiClose(docId, 0))
    F_ApiAlert("Unsaved changes. Can't close.",
              FF_ALERT_CONTINUE_WARN);
. . .
```

Quitting a Frame session

To quit a Frame session, call `F_ApiClose()`, with `Id` set to `FV_SessionId`. For the syntax of `F_ApiClose()`, see “Closing documents and books” on page 260.

For example, to close all the open documents and books in the current Frame session, and quit the session, use the following code:

```
. . .
F_ApiClose(FV_SessionId, FF_CLOSE_MODIFIED);
. . .
```

Because of the flag set to `FF_CLOSE_MODIFIED`, if any books or documents have been changed, the FrameMaker product abandons the changes.

Comparing documents and books

To compare two versions of a document or book using a FrameMaker product’s built-in comparison feature, use `F_ApiCompare()`.

The syntax for `F_ApiCompare()` is:

```
F_CompareRetT F_ApiCompare(F_ObjHandleT olderId,
    F_ObjHandleT newerId,
    IntT flags,
    StringT insertCondTag,
    StringT deleteCondTag,
    StringT replaceText,
    IntT compareThreshold);
```

This argument	Means
<code>olderId</code>	The ID of the older version of the document or book.
<code>newerId</code>	The ID of the newer version of the document or book.
<code>flags</code>	Bit flags that specify how to generate the summary and composite documents.
<code>insertCondTag</code>	The condition tag to apply to insertions shown in the composite document. For no insert condition tag, specify <code>NULL</code> .
<code>deleteCondTag</code>	The condition tag to apply to deletions shown in the composite document. For no delete condition tag, specify <code>NULL</code> .

This argument	Means
replaceText	Text to appear in place of the deleted text. For no replacement text, specify <code>NULL</code> .
compareThreshold	Percentage of words that can change before paragraphs are considered <i>not equal</i> . If two paragraphs are equal, word differences between them are shown within a paragraph in the composite document. If a paragraph is not equal to another, it is marked inserted or deleted. To specify an 85% threshold, set <code>compareThreshold</code> to 85. The default value is 75.

The `F_CompareRetT` structure is defined as:

```
typedef struct {
    F_ObjHandleT sumId; /* ID of the summary document */
    F_ObjHandleT compId; /* ID of the composite document */
} F_CompareRetT;
```

The following values can be ORed into the `flags` argument.

This value	Means
<code>FF_CMP_SUMMARY_ONLY</code>	Generate summary document, but not composite document
<code>FF_CMP_CHANGE_BARS</code>	Turn on change bars in the composite document
<code>FF_CMP_HYPERLINKS</code>	Put hypertext links in the summary document
<code>FF_CMP_SUMKIT</code>	Open the summary document
<code>FF_CMP_COMPKIT</code>	Open the composite document

If you specify the `FF_CMP_SUMKIT` or `FF_CMP_COMPKIT` flags, `F_ApiCompare()` opens the summary and comparison documents and returns their IDs in the `F_CompareRetT` structure. It does *not* make these documents visible to the user. If you want them to be visible, you must set each of the document's `FP_DocIsOnScreen` properties to `True`.

Example

The following code opens two documents and compares them as specified in the dialog boxes shown in Figure 4-2. It makes the summary document visible.

```

. . .
F_ObjHandleT oldId, newId;
IntT flags;
F_CompareRetT cmp;

oldId = F_ApiSimpleOpen("/tmp/1Chapter", False);
newId = F_ApiSimpleOpen("/tmp/1Chapter.new", False);

flags = FF_CMP_CHANGE_BARS | FF_CMP_COMPKIT | FF_CMP_SUMKIT;

cmp = F_ApiCompare(oldId, newId, flags, "Comment",
    "", "Replaced Text", 75);

if (FA_errno != FE_Success)
    F_ApiAlert("Couldn't compare", FF_ALERT_CONTINUE_NOTE);
. . .

```

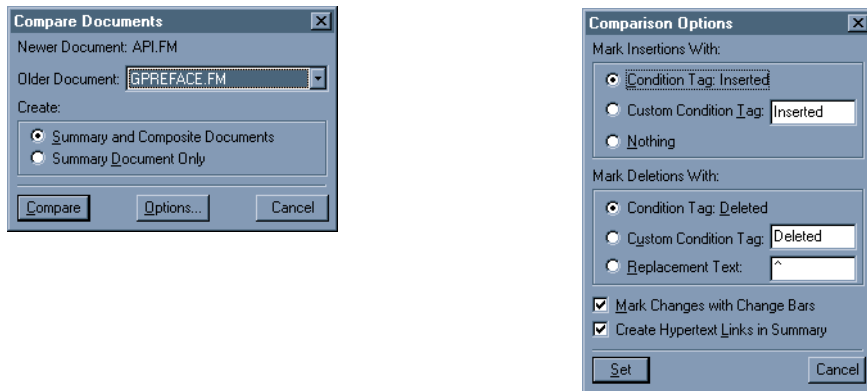


Figure 4-2 Specifications for Compare Documents

Updating and generating documents and books

The API provides a variety of functions that allow you to generate, update, and reformat documents and books. The following sections discuss these functions.

Updating the files in a book

To update the numbering, text insets, cross-references, etc. in all the files in a book, and to programmatically specify responses to warnings and messages that the FrameMaker product issues, use `F_ApiUpdateBook()`. You can specify all aspects of the operation, or you can specify some aspects and allow the user to decide others. For example, you can instruct the FrameMaker product to update view-only files, or to abort the update when it encounters a view-only file.

To use `F_ApiUpdateBook()`, you should first understand property lists and how to manipulate them directly. For more information on this subject, see “Representing object characteristics with properties” on page 65 and “Manipulating property lists directly” on page 295.

The syntax for `F_ApiUpdateBook()` is:

```
ErrorT F_ApiOpen(F_ObjHandleT bookId,
    F_PropValsT *updateParamsp,
    F_PropValsT **updateReturnParamssp);
```

This argument	Means
<code>bookId</code>	The ID of the book you will update.
<code>updateParamsp</code>	A property list (script) that tells the FrameMaker product how to update the book and how to respond to errors and other conditions that arise. Use <code>F_ApiGetUpdateBookDefaultParams()</code> or <code>F_ApiAllocatePropVals()</code> to create and allocate memory for this property list. To use the default list, specify <code>NULL</code> .
<code>updateReturnParamssp</code>	A property list that provides information on how the FrameMaker product updated the book.

.....
IMPORTANT: Always initialize the pointer to the property list that you specify for `openReturnParamssp` to `NULL` before you call `F_ApiUpdateBook()`.

If `F_ApiUpdateBook()` is successful, it returns `FE_Success`. Otherwise, it returns an error which has the same value as `FA_errno`.

To call `F_ApiUpdateBook()`, do the following:

- 1 *Initialize the pointer to the `updateReturnParamspp` property list to `NULL`.*
- 2 *Create an `updateParamspp` property list.*
You can get a default list by calling `F_ApiGetUpdateBookDefaultParams()`, or you can create a list from scratch.
- 3 *Call `F_ApiUpdateBook()`.*
- 4 *Check the Update status.*
Check the returned values in the `updateReturnParamspp` list for the name of the opened file and other information about how the FrameMaker product opened the file.
- 5 *Deallocate memory for the `updateParamspp` and `updateReturnParamspp` property lists.*
Use `F_ApiDeallocatePropVals()` to deallocate memory for the lists.

Generating files for a book

To generate and update files for a book, use `F_ApiSimpleGenerate()`.

The book and its generated files must be set up before you call

`F_ApiSimpleGenerate()`.

The syntax for `F_ApiSimpleGenerate()` is:

```
IntT F_ApiSimpleGenerate(F_ObjHandleT bookId,
    IntT interactive,
    IntT makeVisible);
```

This argument	Means
<code>bookId</code>	The ID of the book for which to generate files
<code>interactive</code>	Specifies whether to display warnings and messages to the user (True displays messages and warnings)
<code>makeVisible</code>	Specifies whether to display generated files (True displays the files)

Importing formats

To import formats from a document to all the documents in a book or from one document to another document, use `F_ApiSimpleImportFormats()`.

The syntax for `F_ApiSimpleImportFormats()` is:

```
IntT F_ApiSimpleImportFormats(F_ObjHandleT bookId,
    F_ObjHandleT fromDocId,
    IntT formatFlags);
```

This argument	Means
<code>bookId</code>	The ID of the book or document to which to import formats
<code>fromDocId</code>	The ID of the document from which to import formats
<code>formatFlags</code>	Bit field that specifies the formats to import

You can OR the values in the following table into the `formatFlags` parameter to specify which formats to import.

This value	Means
<code>FF_UFF_COLOR</code>	Import colors
<code>FF_UFF_COMBINED_FONTS</code>	Import combined font definitions
<code>FF_UFF_COND</code>	Import conditional text settings
<code>FF_UFF_DOCUMENT_PROPS</code>	Import document properties
<code>FF_UFF_FONT</code>	Import Character Catalog formats
<code>FF_UFF_MATH</code>	Import equation settings
<code>FF_UFF_PAGE</code>	Import page layouts
<code>FF_UFF_PGF</code>	Import Paragraph Catalog formats
<code>FF_UFF_REFPAGE</code>	Import reference pages
<code>FF_UFF_TABLE</code>	Import Table Catalog formats
<code>FF_UFF_VAR</code>	Import variable formats
<code>FF_UFF_XREF</code>	Import cross-reference formats
<code>FF_UFF_REMOVE_EXCEPTIONS</code>	Remove exception formats from target documents
<code>FF_UFF_REMOVE_PAGE_BREAKS</code>	Remove all forced page breaks from target documents

Executing other updating and formatting commands

The API provides several functions that allow you to execute FrameMaker product commands that update and reformat entire documents.

The syntax for the functions is:

```
IntT F_ApiClearAllChangebars (F_ObjHandleT docId);
IntT F_ApiRehyphenate (F_ObjHandleT docId);
IntT F_ApiResetReferenceFrames (F_ObjHandleT docId);
IntT F_ApiResetEqnSettings (F_ObjHandleT docId);
IntT F_ApiRestartPgfnNumbering (F_ObjHandleT docId);
IntT F_ApiUpdateVariables (F_ObjHandleT docId);
IntT F_ApiUpdateXRefs (F_ObjHandleT docId,
                      IntT updateXRefFlags);
```

This argument	Means
docId	ID of the document to update or reformat

These functions behave like the corresponding commands in the user interface. They are useful for clients that need to update and reformat multiple files. For more information on a particular function, look it up in the chapter, “FDK Function Reference,” in the FDK Programmer’s Reference.

Example

The following code opens a book and resets the change bars in each of its component documents:

```
. . .
#include "fmemory.h"

F_ObjHandleT bookId, compId, docId;
StringT compName;

bookId = F_ApiSimpleOpen("/tmp/my.book", False);
compId = F_ApiGetId(FV_SessionId, bookId,
                   FP_FirstComponentInBook);

/* Traverse book's components, opening each one
 * and clearing its change bars.
 */
while (compId)
{
    compName = F_ApiGetString(bookId, compId, FP_Name);
    docId = F_ApiSimpleOpen(compName, False);
    F_Free(compName);
    F_ApiClearAllChangebars(docId);
    compId = F_ApiGetId(bookId, compId, FP_NextComponentInBook);
}
. . .
```

Controlling Undo/Redo in the FDK API

Undo/Redo in FrameMaker is controlled by the following:

- Initialization Flag
- Session Properties
- API Functions

Initialization Flag to explicitly enable or disable undo/redo

The `EnableUndoInFDK` flag in the initialization file (`maker.ini`) allows you to explicitly enable or disable undo/redo functionality for API commands, and its associated overhead. It is `false` (off) by default, which means that the undo behavior is the same as in previous releases; that is, calls to API commands clear the undo and redo stacks in the selected document, and API commands cannot be undone. To enable the new undo behavior for API commands, set the flag to `true`. (This flag does not affect the FrameMaker user interface or interactive behavior.)

When `EnableUndoInFDK` is true, all API commands that modify document contents can be undone (see “Undoable API Commands” on page 270). Commands that do not modify content, such as saving a document, copying text, or manipulating windows, cannot be undone and are not recorded in the command history (undo stack).

Session Properties to Control Undo/Redo

FP_UndoFDKRecording - This property, can override the default value specified in the initialization flag `EnableUndoInFDK`. Use `F_ApiSetInt` to set this property value, and `F_ApiGetInt` to retrieve it. Set the property to zero to disable FDK Undo recording for a session, or to a non-zero value to enable Undo recording.

FP_StackWarningLevel - This property determines how warnings are displayed when history-clearing operations occur. It corresponds to an option set in the Preferences dialog, and to the preference-file flag `hpWarning`. Use `F_ApiSetInt` to set this property value, and `F_ApiGetInt` to retrieve it. Allowed values are:

- `FvWarnNever`: Disables warnings for history-clearing operations for the session.
- `FvWarnOnce`: Displays a warning when a particular history-clearing command is issued, but does not warn on subsequent uses of that command.
- `FvWarnAlways`: Displays warnings every time a history-clearing command is issued.

API Functions to Control Undo/Redo

The `F_ApiUndoCancel` command explicitly clears both the undo and redo stacks in a specified document. The other individual API commands do not clear the undo stack.

Many API commands call two or more other API functions. By default, each API call is recorded as a separate undo action in the undo stack of the selected document. To treat a series of API calls as one command, call `F_ApiUndoStartCheckpoint` before the first call and `F_ApiUndoEndCheckpoint` after the last call in the group.

Undoable API Commands

The following API commands are undoable

<code>F_ApiAddCols</code>	<code>F_ApiAddRows</code>
<code>F_ApiAddText</code>	<code>F_ApiApplyPageLayout</code>
<code>F_ApiClear</code>	<code>ApiClearAllChangebars</code>
<code>F_ApiCut</code>	<code>F_ApiDelete</code>
<code>F_ApiDeleteCols</code>	<code>F_ApiDeletePropByName</code>
<code>F_ApiDeleteRows</code>	<code>F_ApiDeleteText</code>

F_ApiDeleteTextInsetContents	F_ApiDeleteUndefinedAttributes
F_ApiDemoteElement	F_ApiImport
F_ApiMergeIntoFirst	F_ApiMergeIntoLast
F_ApiNewAnchoredFormattedObject	F_ApiNewAnchoredObject
F_ApiNewBookComponentInHierarchy	F_ApiNewElement
F_ApiNewElementInHierarchy	F_ApiNewGraphicObject
F_ApiNewNamedObject	F_ApiNewSeriesObject
F_ApiNewSubObject	F_ApiNewTable
F_ApiPaste	F_ApiPromoteElement
F_ApiReformat	F_ApiUnStraddleCells
F_ApiResetEqnSettings	F_ApiResetReferenceFrames
F_ApiRestartPgfnNumbering	F_ApiSetAttributeDefs
F_ApiSetAttributes	F_ApiSetElementRange
F_ApiSetId	F_ApiSetInt
F_ApiSetIntByName	F_ApiSetInts
F_ApiSetMetric	F_ApiSetMetricByName
F_ApiSetMetrics	F_ApiSetPoints
F_ApiSetProps	F_ApiSetPropVal
F_ApiSetString	F_ApiSetStrings
F_ApiSetTabs	F_ApiSetTextLoc
F_ApiSetTextProps	F_ApiSetTextPropVal
F_ApiSetTextRange	F_ApiSetTextVal
F_ApiSetUBytesByName	F_ApiUnWrapElement
F_ApiSimpleImportElementDefs	F_ApiSimpleImportFormats
F_ApiSplitElement	F_ApiStraddleCells
F_ApiUnWrapElement	F_ApiUpdateTextInset
F_ApiSave	F_ApiSimpleSave

Simulating user input

To simulate user input, call the API function `F_ApiFcodes()`.

`F_ApiFcodes()` sends an array of function codes (f-codes) to the FrameMaker product. F-codes are hexadecimal codes that specify individual user actions, such as cursor movement and text entry. They are especially useful for manipulating windows. For example, the f-code `KBD_EXPOSEWIN` brings the active document or book window to the front. When you use `F_ApiFcodes()` to send an array of f-codes to a FrameMaker product, it executes each f-code as if the user performed the action.

.....
IMPORTANT: `F_ApiFcodes()` does not work with dialog boxes on Windows.

The syntax for `F_ApiFcodes()` is:

```
IntT F_ApiFcodes(IntT len,
                IntT *vec);
```

This argument	Means
<code>len</code>	The length of the array of f-codes in bytes
<code>vec</code>	The array of f-codes to send to the FrameMaker product

The following table lists some user actions and the f-codes that emulate them.

User action	F-code
Move insertion point to first flow on the current page	<code>CSR_TOP</code>
Move insertion point to the start of the next word	<code>CSR_NEXT_BOW</code>
Move insertion point to the beginning of the next paragraph	<code>CSR_NEXT_BOP</code>
Center the current paragraph	<code>PGF_CENTER</code>
Left justify the current paragraph	<code>PGF_LEFT</code>
Make selected text bold	<code>TXT_BOLD</code>

For a complete list of f-codes, see the `fcodes.h` file shipped with the FDK.

`F_ApiFcodes()` uses the current focus in a dialog box or a visible document. If you want to execute a set of f-codes in a particular dialog box or document, make sure that the dialog box or document is active. To make a dialog box active, use f-codes such as

FOCUS_INPUT_SEARCH and FOCUS_INPUT_PGFFMT. To make a document active, set the session property FP_ActiveDoc to the document's ID.

Many f-codes perform tasks that API functions also perform. Whenever possible, try to use the other API functions instead of F_ApiFcodes() to perform these tasks. F_ApiFcodes() does not provide error or status feedback for individual f-codes, whereas each API function stores an error code to FA_errno when it fails. It is also difficult to debug lengthy f-code sequences.

The following code uses f-codes to enter the string HI!, select the text, and then make it bold:

```

. . .
static IntT fcodes[] = {CSR_TOP, 'H', 'I', '!', HIGH_WORD_PREV,
                        TXT_BOLD};
F_ApiFcodes(sizeof(fcodes)/sizeof(IntT), fcodes);
. . .

```

Straddling table cells

To straddle and unstraddle table cells, use F_ApiStraddleCells() and F_ApiUnStraddleCells().

The syntax for these functions is:

```

IntT F_ApiStraddleCells(F_ObjHandleT docId,
    F_ObjHandleT cellId,
    IntT heightInRows,
    IntT widthInCols);

IntT F_ApiUnStraddleCells(F_ObjHandleT docId,
    F_ObjHandleT cellId,
    IntT heightInRows,
    IntT widthInCols);

```

This argument	Means
docId	The ID of the document containing the table
cellId	The ID of the first (leftmost and uppermost) cell to straddle or unstraddle
heightInRows	The number of cells to straddle or unstraddle vertically
widthInCols	The number of cells to straddle or unstraddle horizontally

Both heightInRows and widthInCols must be greater than 0. At least one of them must be greater than 1. The cells you straddle must all be from the same type of row. You can't, for example, straddle a set of cells that are in both heading and body

rows. You also can't straddle cells that are already straddled. If the cells you specify include cells that are already straddled, `F_ApiStraddleCells()` returns `FE_BadOperation`.

When you or the user straddle table cells, the FrameMaker product does *not* delete any of the `FO_Cell` objects that represent the cells. It links the paragraphs from the straddled cells into a single list. The `FP_FirstPgf` and `FP_LastPgf` properties of each cell in the straddle specify the first and last paragraphs in this list.

Example

The following code straddles the first two cells in the first column of a table:

```
. . .
F_ObjHandleT docId, tableId, firstrowId, cellId;

/* Get IDs of document, table, first row, and first cell. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tableId = F_ApiGetId(FV_SessionId, docId, FP_FirstTblInDoc);
firstrowId = F_ApiGetId(docId, tableId, FP_FirstRowInTbl);
cellId = F_ApiGetId(docId, firstrowId, FP_FirstCellInRow);

/* If there are two cells in the row, straddle them. */
if (F_ApiGetInt(docId, tableId, FP_TblNumCols) < 2)
    F_ApiAlert("Not enough columns!", FF_ALERT_CONTINUE_NOTE);
else F_ApiStraddleCells(docId, cellId, 1, 2);
. . .
```

Executing FrameMaker commands

The following sections describe how to programmatically execute FrameMaker commands.

Manipulating elements

The API provides several functions that allow you to execute basic commands that manipulate elements.

The syntax for these functions is:

```
VoidT F_ApiDemoteElement (F_ObjHandleT docId);  
VoidT F_ApiMergeIntoFirst (F_ObjHandleT docId);  
VoidT F_ApiMergeIntoLast (F_ObjHandleT docId);  
VoidT F_ApiPromoteElement (F_ObjHandleT docId);  
VoidT F_ApiSplitElement (F_ObjHandleT docId);  
VoidT F_ApiUnWrapElement (F_ObjHandleT docId);  
VoidT F_ApiWrapElement (F_ObjHandleT docId,  
                        F_ObjHandleT edefId);
```

This argument	Means
docId	ID of the document containing selected text and/or structure elements
edefId	ID of element definition for the new element

These functions behave like the corresponding commands in the user interface. They all use the current text or element selection in the specified document to determine which text and elements to manipulate. You can allow the user to set the text or element selection, or you can do it programmatically. For information on setting the text selection programmatically, see “Getting and setting the insertion point or text selection” on page 323. For more information on setting the element selection programmatically, see “Getting and setting the structural element selection” on page 331

Importing element definitions into FrameMaker documents and books

To import element definitions from a FrameMaker document or book to a FrameMaker document or book, use `F_ApiSimpleImportElementDefs()`.

The syntax for `F_ApiSimpleImportElementDefs()` is:

```
IntT F_ApiSimpleImportElementDefs (
    F_ObjHandleT docOrBookId,
    F_ObjHandleT fromDocOrBookId,
    IntT importFlags);
```

This argument	Means
<code>docOrBookId</code>	The ID of the document or book to import element definitions to.
<code>fromDocOrBookId</code>	The ID of the document or book from which to import element definitions.
<code>importFlags</code>	See the following table for the flags that you can OR into this parameter.

The following table lists flags that you can OR into the `importFlags` parameter:

Flag	Meaning
<code>FF_IED_REMOVE_OVERRIDES</code>	Clear format overrides.
<code>FF_IED_REMOVE_BOOK_INFO</code>	If <code>docOrBookId</code> specifies a document, clear formatting inherited from the parent book.
<code>FF_IED_DO_NOT_IMPORT_EDD</code>	If the document specified by <code>fromDocOrBookId</code> is an EDD, don't treat it as an EDD; just import its element catalog.
<code>FF_IED_NO_NOTIFY</code>	Do not issue the <code>FA_Note_PreImportElemDefs</code> or <code>FA_Note_PostImportElemDefs</code> notifications.

If you import element definitions to a book, `F_ApiSimpleImportElementDefs()` imports element definitions to each book component for which the `FP_ImportFmtInclude` property is set to `True`.

Calling FrameMaker clients programmatically

Much of the structured document functionality FrameMaker provides is implemented in FDK clients. To call this functionality programmatically, you must use `F_ApiCallClient()`.

`F_ApiCallClient()` requires you to specify a client's registered name and a string, which it passes to the client. The following table lists FrameMaker functionality and the registered names of the clients you can call to invoke it programmatically.

Functionality	Registered client name
Element catalog manager	Element Catalog Manager
Structure generator	Structure Generator
Reading and writing Structured documents and reading, writing, and updating DTD and EDD documents	FmDispatcher

The following table lists the strings you pass to the structure generator client to programmatically generate structure in a document or book.

String	Meaning
<code>INPUTDOCID</code> <i>objectID</i>	The ID of the input document or book.
<code>RULEDOCID</code> <i>objectID</i>	The ID of the rule table document.
<code>OUTPUTDOCNAME</code> <i>filename</i>	The full pathname of the output document or book. This string is optional. If you do not specify a pathname, the structure generator leaves the document unsaved and open.
<code>LOGNAME</code> <i>filename</i>	The full pathname of a log file. This string is optional. If you do not specify a pathname, the structure generator leaves the log file unsaved and open.
<code>StructureDoc</code>	Instructs the structure generator to generate structure, using the strings listed above.

To programmatically generate structure for a document or a book, you call `F_ApiCallClient()` multiple times, each time passing it one of the strings listed in the table above. For example, the following code generates structure for a document:

```
. . .
F_ObjHandleT inputDocId, ruleTblDocId;
UCharT buf[64];
. . .
F_Sprintf(buf, "INPUTDOCID %d", inputDocId);
F_ApiCallClient("StructGen", buf);

F_Sprintf(buf, "RULEDOCID %d", ruleTblDocId);
F_ApiCallClient("StructGen", buf);

F_ApiCallClient("StructGen", "OUTPUTDOCNAME /tmp/mystruct.doc");
F_ApiCallClient("StructGen", "LOGNAME /tmp/logfile.doc");
F_ApiCallClient("StructGen", "StructureDoc");
. . .
```

Note that all of the documents you specify must be open before you call the structure generator. If you are generating structure for a large number of documents, you can greatly speed processing by opening the documents invisibly. To open a document invisibly, set the `FS_MakeVisible` property of the Open script to `False`.

For a complete list of the strings you can pass to the structure generator and other FrameMaker clients, see “`F_ApiCallClient()`” in the FDK Programmer’s Reference guide.

Getting and Setting Properties

This chapter describes how to make changes in a FrameMaker product session, book, or document by getting and setting property values. It discusses how to get and set individual properties and entire property lists. It also provides some tips for getting and setting the properties of specific types of objects.

What you can do with object properties

In the FrameMaker product user interface, the user can change an object in a variety of ways. For example, the user can change the size and fill pattern of a graphic object or the starting page number of a book component.

Each API object has a *property list*, a set of properties describing its attributes. Your API client can do anything a user can do to an object by getting and setting the properties in the object's property list. For example, your client can set properties to:

- Change a graphic object's size, fill pattern, or position in the back-to-front order
- Make a document or book active
- Change a book component's position in a book
- Change a paragraph's format

Your client can also change properties that the user doesn't have access to. For example, your client can set properties to:

- Make a document or book visible or invisible
- Keep the FrameMaker product from reformatting a document every time a change is made

The API ensures that your client doesn't corrupt a document by setting properties to illegal values. When you change a property, the API also automatically changes other properties as needed to preserve the integrity of the document or book.

There are a number of read-only properties that you can get but not set. For a complete list of object properties and their possible values, see chapter, "Object Reference," in the FDK Programmer's Reference.

To change a session, document, or book by setting object properties, follow these general steps:

1 *Find out which objects represent the things you want to change.*

To change something in a session, book, or document, you need to know which objects the API uses to represent it. For a description of how the API uses objects to represent things in FrameMaker products, see Part II, “Frame Product Architecture.”

2 *Get the IDs of the objects you want to change.*

To set an object’s properties, you must specify its ID. The API provides functions for retrieving object IDs.

3 *Manipulate the objects’ properties.*

The API provides functions for getting and setting individual properties and entire property lists.

For example, the API represents a FrameMaker product session with an `FO_Session` object. You don’t need to get a session’s ID, because there is only one session and its ID is always `FV_SessionId`. To find all the session characteristics you can change, look up “Session” in the chapter, “Object Reference,” in the *FDK Programmer’s Reference*.

You can, for example, change the session’s automatic save time. The API represents the automatic save time with an integer (`IntT`) property named `FP_AutoSaveSeconds`. To set it to 60 seconds, use the following code:

```
F_ApiSetInt(0,           /* Sessions have no parent */
            FV_SessionId, /* The session’s ID */
            FP_AutoSaveSeconds, /* The property to set */
            60);          /* The value to set it to */
```

The following sections describe steps 2 and 3 in greater detail.

Getting the IDs of the objects you want to change

Every object in a session has an ID. To get or set the properties of a particular object, you must specify its ID. In Frame book and document architecture, objects are organized in linked lists: an object has properties that specify the IDs of other objects, which have properties that specify the IDs of other objects, and so on. To get the IDs of specific objects, you traverse the linked lists by querying these properties. For diagrams and descriptions of the linked lists in Frame architecture, see Part II, “Frame Product Architecture.”

To query a property that specifies an object ID, use `F_ApiGetId()`, which is defined as:

```
F_ObjHandleT F_ApiGetId(F_ObjHandleT docId,  
                        F_ObjHandleT objId,  
                        IntT propNum);
```

This argument	Means
<code>docId</code>	The ID of the document, book, or session containing the object whose property you want to query.
<code>objId</code>	The ID of the object whose property you want to query.
<code>propNum</code>	The property to query. Specify one of the API-defined constants, such as <code>FP_ActiveDoc</code> .

`F_ApiGetId()` returns the ID specified by the property. If the property doesn't specify an ID or an error occurs, `F_ApiGetId()` returns 0.

To get an object's ID, you start traversing at the object that represents the session (the `FO_Session` object), because it is the only object whose ID (`FV_SessionId`) you know from the start.

From the `FO_Session` object, you can get the IDs of the active and open documents and books in the session. `FO_Session` objects have properties, named `FP_ActiveDoc` and `FP_ActiveBook`, that specify the IDs of the active document or book. A document or a book is active if it has input focus.

`FO_Session` objects also have properties, named `FP_FirstOpenDoc` and `FP_FirstOpenBook`, that specify the first document and the first book in the linked lists of open documents and books in a session. `FO_Doc` objects have a property named `FP_NextOpenDocInSession` that specifies the ID of the next `FO_Doc` object in the list of open documents. `FO_Book` objects have a property named `FP_NextOpenBookInSession` that specifies the ID of the next `FO_Book` object in the list of open books. If an `FO_Doc` or an `FO_Book` object is the last object in the list, its `FP_NextOpenDocInSession` or `FP_NextOpenBookInSession` property is set to 0. For a diagram of how the API represents the documents and books in a session, see Figure 1-2 on page 71.

Suppose you want to display the IDs of the active document and all the open documents in a session. You can use the following code to do this:

```

. . .
#include "futils.h"
F_ObjHandleT docId;
UCharT msg[256];

/* Get the ID of the active document and display it. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
if (docId)
    {
    F_Sprintf(msg, "The active document's ID is 0x%x.", docId);
    F_ApiAlert(msg, FF_ALERT_CONTINUE_NOTE);
    }

/* Get ID of the first document in list of open documents. */
docId = F_ApiGetId(0, FV_SessionId, FP_FirstOpenDoc);

/* Traverse list of open documents and display their IDs. */
while (docId)
    {
    F_Sprintf(msg, "The document's ID is 0x%x.", docId);
    F_ApiAlert(msg, FF_ALERT_CONTINUE_NOTE);
    docId = F_ApiGetId(FV_SessionId, docId,
                      FP_NextOpenDocInSession);
    }
. . .

```

This code displays the ID of the active document twice, because the active document is included in the list of open documents.

The linked list of open documents in a session isn't in any particular order. The first document in the list is *not* necessarily the active document or the first document that was opened.

Another way to get a document ID is to use `F_ApiSimpleOpen()`, `F_ApiOpen()`, or `F_ApiSimpleNewDoc()` to open or create the document. These functions all return the IDs of the document they open or create.

Traversing lists of objects in a document

Once you have the ID of a document, you can query its properties to get to the lists of objects that it contains. The document has a number of properties that point to these lists. For example, the document's `FP_FirstGraphicInDoc` property specifies the ID of the first graphic object in the list of its graphic objects and its `FP_FirstBodyPageInDoc` property specifies the first body page in the list of its body pages. Except for the lists of pages, the lists are completely unordered. For example, the first graphic object in the list of graphic objects is not necessarily the first graphic that appears in the document.

Suppose you want to traverse the list of all the paragraphs in the active document. To do this, you can use the following code:

```
. . .
#include "futils.h"
F_ObjHandleT docId, pgfId;
UCharT msg[256];

/* Get the ID of the active document. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);

/* Get ID of the first paragraph in the list of paragraphs. */
pgfId = F_ApiGetId(FV_SessionId, docId, FP_FirstPgfInDoc);

/* Traverse the list of paragraphs and display their IDs.
** Even empty documents have several paragraphs, because text
** columns on master pages contain paragraphs.
*/
while (pgfId)
{
    F_Sprintf(msg, "The paragraph's ID is 0x%x.", pgfId);
    F_ApiAlert(msg, FF_ALERT_CONTINUE_NOTE);
    pgfId = F_ApiGetId(docId, pgfId, FP_NextPgfInDoc);
}
. . .
```

The paragraphs in the list are not ordered.

Traversing lists of graphic objects

The API does not maintain separate lists of the different types of graphic objects in a document. For example, a document's text columns (FO_TextFrame objects), rectangles (FO_Rectangle objects), and anchored frames (FO_AFrame objects) are all in the same list. To determine objects' types as you traverse them, use `F_ApiGetObjectTypes()`.

The syntax for `F_ApiGetObjectTypes()` is:

```
UIntT F_ApiGetObjectTypes(F_ObjHandleT docId,
                          F_ObjHandleT objId);
```

This argument	Means
<code>docId</code>	The ID of the document, book, or session containing the object
<code>objId</code>	The ID of the object whose type you want to get

For example, the following code counts the number of anchored frames in the active document:

```
. . .
#include "futils.h"
IntT numFrames = 0;
F_ObjHandleT docId, objId;
UCharT msg[256];

docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);

/* Get ID of first graphic in the list of graphics. */
objId = F_ApiGetId(FV_SessionId, docId, FP_FirstGraphicInDoc);

/* Traverse list of graphics, counting anchored frames. */
while (objId)
{
    if (F_ApiGetObjectTypes(docId,objId) == FO_AFrame) numFrames++;
    objId = F_ApiGetId(docId, objId, FP_NextGraphicInDoc);
}

F_Printf(msg, "The document has %d anchored frames.",
        numFrames);
F_ApiAlert(msg, FF_ALERT_CONTINUE_NOTE);

. . .
```

Traversing ordered lists of objects

Traversing the list of all the objects of a certain type in a document is useful if you want to get every object of that type and the order doesn't matter to you. However, it isn't very useful if you want the objects in some kind of order, such as the order in which they appear on a document's pages. To get objects in order, you must traverse the ordered lists that the API maintains. There are ordered lists of the graphic objects in a frame, the text columns within a flow, and many other objects. These lists can be deeply nested, for example, when a frame contains a frame that contains some graphic objects.

There are a variety of object properties you can query to get to ordered lists. For example, to get to the list of graphic objects in a frame, you can query the frame's `FP_FirstGraphicInFrame` or `FP_LastGraphicInFrame` properties. If you already have one of the graphic object's IDs, you can query its `FP_PrevGraphicInFrame` and `FP_NextGraphicInFrame` properties to get to the objects behind it and in front of it in the list. The order of the list corresponds to the back-to-front order of the graphics in the frame. For information on the linked lists that a particular object is included in, see the section that discusses that object in Chapter 2, "Frame Document Architecture."

Although there are ordered lists of the paragraphs within each of a document's flows, there is no ordered list of flows. You can get the paragraphs only in the order in which they occur within an individual flow.

To get the paragraphs within an individual flow in order, you navigate from the flow to the first text frame in the flow, to the first paragraph in that text frame. For example, to get the paragraphs in a document's main flow in order, you can use the following code:

```
. . .
#include "futils.h"
F_ObjHandleT docId, pgfId, flowId, textFrameId;
UCharT msg[256];

docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);

/* Get ID of main flow, then the first text column in the
 * flow, then the first paragraph in that text column.
 */
flowId = F_ApiGetId(FV_SessionId, docId, FP_MainFlowInDoc);
textFrameId = F_ApiGetId(docId, flowId,
                        FP_FirstTextFrameInFlow);
pgfId = F_ApiGetId(docId, textFrameId, FP_FirstPgf);

/* Traverse ordered list of paragraphs in the flow. */
while (pgfId)
    {
        F_Printf(msg, "The paragraph's ID is 0x%x.", pgfId);
        F_ApiAlert(msg, FF_ALERT_CONTINUE_NOTE);
        pgfId = F_ApiGetId(docId, pgfId, FP_NextPgfInFlow);
    }
. . .
```

For a diagram of the links between flows, text frames, and paragraphs, see “The list of paragraphs in a flow” on page 106.

Getting the IDs of selected objects

Document objects have properties that allow you to get the IDs of the following types of selected objects:

- Graphic objects
- Tables and table rows

To get the IDs of selected structural elements in FrameMaker documents, you must call a special function, `F_ApiGetElementRange()`.

For background information on selection in Frame documents, see “How the API represents the selection in a document” on page 82. For information on getting selected text, see “Getting and setting the insertion point or text selection” on page 323.

Getting the IDs of selected graphic objects

The API maintains an unordered list of all the selected graphic objects in a document. To manipulate graphic objects the user has selected, you traverse this list. For example, the following code sets the fill pattern of all the selected graphic objects in the active document to black:

```
. . .
F_ObjHandleT docId, objId;

docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
objId = F_ApiGetId(FV_SessionId, docId,
                  FP_FirstSelectedGraphicInDoc);
while (objId)
    {
    F_ApiSetInt(docId, objId, FP_Fill, FV_FILL_BLACK);
    objId = F_ApiGetId(docId, objId,
                      FP_NextSelectedGraphicInDoc);
    }
. . .
```

Getting the IDs of selected tables and table rows

If a table is selected, you can get its ID by querying the document property, `FP_SelectedTbl`. A table is considered selected if any of its cells are selected or the insertion point is in any of its cells.

To get the IDs of the rows selected within a table, query the `FO_Tbl` object’s `FP_TopRowSelection` and `FP_BottomRowSelection` properties. `FP_TopRowSelection` specifies the ID of the row at the top of the selection; `FP_BottomRowSelection` specifies the ID of the row at the bottom of the selection.

To determine which cells in a row are selected, query a table’s `FP_LeftColNum` and `FP_RightColNum` properties. `FP_LeftColNum` specifies the number (starting from 0) of the leftmost selected column; `FP_RightColNum` specifies the number of the rightmost selected column.

If a range of text that includes several tables is selected, and you want to get the tables’ IDs, you must get the text selection and traverse all the table anchor text items in it. For more information on getting the text selection, see “Getting and setting the insertion point or text selection” on page 323.

Getting the IDs of selected structural elements

For information on getting the IDs of selected structural elements, see “Getting and setting the structural element selection” on page 331.

Getting the IDs of formats and other named objects

The following are some of the types of objects that are *named* (identified by a unique name).

- `FO_CharFmt`
- `FO_Color`
- `FO_CombinedFontDfn`
- `FO_Command`
- `FO_CondFmt`
- `FO_ElementDef`
- `FO_Flow`
- `FO_FmtChangeList`
- `FO_MarkerType`
- `FO_MasterPage`
- `FO_Menu`
- `FO_PgfFmt`
- `FO_TblFmt`
- `FO_UnanchoredFrame` (named frames on reference pages)
- `FO_VarFmt`
- `FO_XRefFmt`

The API maintains all the named objects of a particular type in a linked list. To get the objects, you can query `FO_Doc` properties and traverse the list. For example, to get all the variable formats in a document, query the `FO_Doc` object’s `FP_FirstVarFmtInDoc` property, and then traverse the `FP_NextVarFmtInDoc` properties from one `FO_VarFmt` object to the next.

If you only want the ID for a single named object, it is usually easier to use `F_ApiGetNamedObject()`. The syntax for `F_ApiGetNamedObject()` is:

```
F_ObjHandleT F_ApiGetNamedObject(F_ObjHandleT docId,  
    IntT objType,  
    StringT name);
```

This argument	Means
<code>docId</code>	The ID of the document or book containing the object
<code>objType</code>	The type of object (for example, <code>FO_VarFmt</code>)
<code>name</code>	The name of the object for which to get the ID

For example, the following code gets the ID of the Paragraph Catalog format named `Body` in the active document:

```
. . .  
F_ObjHandleT docId, pgfFmtId;  
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);  
pgfFmtId = F_ApiGetNamedObject(docId, FO_PgfFmt, "Body");  
. . .
```

Getting IDs for objects that have persistent identifiers

FrameMaker products assign a persistent *unique identifier* (UID) to each object in a document that isn't identified by a name. The UID, unlike the object's ID, does not change from one session to another. No matter how many times you open and exit a document, an object's UID remains the same.

It is important to note that cut and paste functions will assign new UID's to the text. For example conditionalizing text will change a paragraph's UID.

To get an object's UID, you query its `FP_Unique` property with `F_ApiGetInt()`. If you already know an object's UID, you can find its ID in the current session by calling `F_ApiGetUniqueObject()`.

The syntax for `F_ApiGetUniqueObject()` is:

```
F_ObjHandleT F_ApiGetUniqueObject(F_ObjHandleT docId,  
    IntT objType  
    IntT unique);
```

This argument	Means
<code>docId</code>	The ID of the document containing the object
<code>objType</code>	The object type (for example <code>FO_Pgf</code>)
<code>unique</code>	The object's UID

For an example of how you can use UIDs, see “`F_ApiGetUniqueObject()`” in the FDK Programmer's Reference guide.

Manipulating properties

The API allows you to get and set either an individual property or a property list for an object. It is generally easier to get and set properties individually. However, some tasks, such as applying a Paragraph Catalog format to a paragraph, are easier to perform by getting and setting property lists.

Getting and setting individual properties

To get or set an individual property, use the `F_ApiGetPropertyType()` or `F_ApiSetPropertyType()` function that corresponds to the property's data type. For example, to get an integer, enum, or boolean (`IntT` data type) property, use `F_ApiGetInt()`. To set a property that represents a set of strings (`F_StringsT` data type), use `F_ApiSetStrings()`. The data types of API properties are listed in the chapter, "Object Reference," in the FDK Programmer's Reference. The following table lists the functions you use to set different types of properties.

The API also provides special functions to get and set properties that are identified by names. These functions are used for getting and setting inset properties only. They are discussed in Chapter 12, "Using Imported Files and Insets."

Property's data type	Functions to get and set property
<code>F_AttributesT</code>	<code>F_ApiGetAttributes()</code> <code>F_ApiSetAttributes()</code>
<code>F_AttributeDefsT</code>	<code>F_ApiGetAttributeDefs()</code> <code>F_ApiSetAttributeDefs()</code>
<code>F_ElementCatalogEntriesT</code>	<code>F_ApiGetElementCatalog()</code>
<code>F_ElementFmtstT</code>	<code>F_ApiGetElementFormats()</code> <code>F_ApiSetElementFormats()</code>
<code>F_ElementRangeT</code>	<code>F_ApiGetElementRange()</code> <code>F_ApiSetElementRange()</code>
<code>F_ObjHandleT</code>	<code>F_ApiGetId()</code> <code>F_ApiSetId()</code>
<code>IntT</code> (including boolean, enum, and ordinal)	<code>F_ApiGetInt()</code> <code>F_ApiSetInt()</code>
<code>F_IntstT</code>	<code>F_ApiGetInts()</code> <code>F_ApiSetInts()</code>
<code>MetricT</code>	<code>F_ApiGetMetric()</code> <code>F_ApiSetMetric()</code>
<code>F_MetricstT</code>	<code>F_ApiGetMetrics()</code> <code>F_ApiSetMetrics()</code>
<code>F_PointstT</code>	<code>F_ApiGetPoints()</code> <code>F_ApiSetPoints()</code>
<code>StringT</code>	<code>F_ApiGetString()</code> <code>F_ApiSetString()</code>

Property's data type	Functions to get and set property
F_StringsT	F_ApiGetStrings() F_ApiSetStrings()
F_TabsT	F_ApiGetTabs() F_ApiSetTabs()
F_TextLocT	F_ApiGetTextLoc() F_ApiSetTextLoc()
F_TextRangeT	F_ApiGetTextRange() F_ApiSetTextRange()

The syntax for most `F_ApiGetPropertyType()` and `F_ApiSetPropertyType()` functions is similar. For example, the syntax for `F_ApiGetInt()` is:

```
IntT F_ApiGetInt(F_ObjHandleT docId,
                F_ObjHandleT objId,
                IntT propNum);
```

This argument	Means
docId	The ID of the document, book, or session containing the object. If the object is a session, specify 0.
objId	The ID of the object whose property you want to query.
propNum	The property to query (for example, <code>FP_FnNum</code>).

The syntax for `F_ApiSetString()` is:

```
VoidT F_ApiSetString(F_ObjHandleT docId,
                    F_ObjHandleT objId,
                    IntT propNum,
                    StringT setVal);
```

This argument	Means
docId	The ID of the document, book, or session containing the object
objId	The ID of the object whose property you want to set
propNum	The property to set, for example, <code>FP_PrintFileName</code>
setVal	The string to which to set the property

You can look up the exact syntax of an `F_ApiGetPropertyType()` or `F_ApiSetPropertyType()` function in the chapter, “FDK Function Reference,” in the FDK Programmer’s Reference.

Suppose you want your client to change some characteristics of the Heading1 paragraph format. To find out how the API represents paragraph formats, look up paragraph formats in Part II, “Frame Product Architecture.” For a complete list of paragraph format properties, see the chapter, “Object Reference,” in the FDK Programmer’s Reference.

The following code demonstrates how to change different types of paragraph format properties:

```
. . .  
#define in (MetricT) (72 * 65536) /* A Frame metric inch */  
F_ObjHandleT docId, pgfFmtId;  
  
/* Get the ID of Heading1 format in active document. */  
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);  
pgfFmtId = F_ApiGetNamedObject(docId, FO_PgfFmt, "Heading1");  
  
/* Set Next Pgf Tag to Heading2. */  
F_ApiSetString(docId, pgfFmtId, FP_NextTag, "Heading2");  
  
/* Turn on Keep With Next. */  
F_ApiSetInt(docId, pgfFmtId, FP_KeepWithNext, True);  
  
/* Set the left indent to 1 inch. */  
F_ApiSetMetric(docId, pgfFmtId, FP_LeftIndent, in);  
. . .
```

This code changes only the Heading1 Paragraph Catalog format. It does not change the formats of paragraphs that have already been tagged with Heading1.

Getting and setting property lists

Because most objects have relatively long property lists, it is often easier to get and set individual properties. However, to perform the following types of tasks, you may need to get and set entire property lists:

- Getting and setting text properties
- Applying table, paragraph, and character formats
- Copying graphic object properties

To get and set property lists, you need to understand how the API represents them. For more information, see “Property lists” on page 66.

`F_ApiGetProps()` and `F_ApiSetProps()` make it easy to get and set property lists.

The syntax for these functions is:

```
F_PropValsT F_ApiGetProps(F_ObjHandleT docId,
                          F_ObjHandleT objId);
```

```
VoidT F_ApiSetProps(F_ObjHandleT docId,
                   F_ObjHandleT objId,
                   F_PropValsT *setVal);
```

This argument	Means
<code>docId</code>	The ID of the session, book, or document containing the object
<code>objId</code>	The ID of the object to get or set the property list for
<code>setVal</code>	The property list to apply to the object

The `F_PropValsT` structure returned by `F_ApiGetProps()` references memory that is allocated by the API. Use `F_ApiDeallocatePropVals()` to free this memory when you are done with it. If `F_ApiGetProps()` fails, the API sets the `len` field of the returned structure to 0.

Example

The following code copies the properties from one selected graphic object to another:

```
. . .
F_PropValsT props;
F_ObjHandleT obj1Id, obj2Id, docId;

/* Get ID of active document and the two selected objects. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
obj1Id = F_ApiGetId(FV_SessionId, docId,
                   FP_FirstSelectedGraphicInDoc);
obj2Id = F_ApiGetId(docId, obj1Id, FP_NextSelectedGraphicInDoc);
/* Make sure two objects are selected, then copy properties. */
if (!(obj1Id && obj2Id)) F_ApiAlert("First select two objects.",
                                   FF_ALERT_CONTINUE_WARN);
else
{
    props = F_ApiGetProps(docId, obj1Id);
    if(props.len == 0) return; /* Get props failed. */
    F_ApiSetProps(docId, obj2Id, &props);
}
. . .
```

Because a graphic object's *x* and *y* coordinates are included in its property list, this code moves the two graphic objects to the same location, with one object overlaying the other.

After you have copied a property list to an object, you can customize the list by changing individual properties.

Manipulating property lists directly

If you are setting individual text properties or using scriptable functions, such as `F_ApiOpen()`, you need to manipulate property lists directly.

The order of the properties in property lists is not guaranteed to remain the same in future versions of FrameMaker products and the Frame API. So, to get a particular property in a list, you must traverse the entire property list and check each property's identifier until you find it. The API provides a convenience routine named `F_ApiGetPropIndex()` that does this for you.

The syntax for `F_ApiGetPropIndex()` is:

```
IntT F_ApiGetPropIndex(F_PropValsT *pvp,
    IntT propNum);
```

This argument	Means
<code>pvp</code>	The property list
<code>propNum</code>	The property whose index you want to get

`F_ApiGetPropIndex()` returns the index of the `F_PropValT` structure that represents the property's property-value pair. If you specify an invalid property for `propNum`, `F_ApiGetPropIndex()` returns `FE_BadPropNum`.

Suppose you want to display the session property that provides the name of the current `FrameMaker` product. The easy way to do this would be to use the following code:

```
. . .
StringT productName;

productName = F_ApiGetString(0, FV_SessionId, FP_ProductName);
F_ApiAlert(productName, FF_ALERT_CONTINUE_NOTE);
. . .
```

To do the same thing by getting the property list for the session and accessing the property directly, use the following code:

```
. . .
IntT i;
F_PropValsT props;

props = F_ApiGetProps(0, FV_SessionId);

i = F_ApiGetPropIndex(&props, FP_ProductName);
F_ApiAlert(props.val[i].propVal.u.sval,
    FF_ALERT_CONTINUE_NOTE);
. . .
```

Allocating and deallocating memory for properties

The `F_ApiGetPropertyType()` functions that return pointers to arrays make copies of the arrays, allocating memory for them. For example, `F_ApiGetString()` does not return a pointer to the actual string used by the `FrameMaker` product. Instead, it creates a copy of the string and returns a pointer to the copy. The API does not

deallocate memory used by the copy of the string. When you are done with it, you must deallocate it.

Similarly, when you call a `F_Api SetPropertyType()` function such as `F_Api SetString()`, the function does not set a pointer to the string you pass to it. Instead it copies the string. The API does not deallocate the string you pass. When you are done with it, you must deallocate it.

For example, the following code queries and displays the `FP_OpenDir` property. It uses the FDE function, `F_Free()`, to free the returned string.

```
. . .
#include "fstrings.h"
#include "fmemory.h"
StringT openDir;

openDir = F_ApiGetString(0, FV_SessionId, FP_OpenDir);
F_ApiAlert(openDir, FF_ALERT_CONTINUE_NOTE);
F_Free(openDir);
. . .
```

For more information on FDE functions, see Part III, “Frame Development Environment (FDE).”

Some API functions return structures containing pointers to arrays. The API allocates memory for these arrays. When you are done with this memory, you must deallocate it. The API provides convenience functions, such as `F_ApiDeallocatePropVals()`, `F_ApiDeallocateStrings()`, and `F_ApiDeallocateMetrics()`, which you can use for this.

For example, to get the property list for an object and then deallocate it, use code similar to the following:

```
. . .
F_PropValsT props;
F_ObjHandleT objId, docId;

props = F_ApiGetProps(docId, objId);
. . .
F_ApiDeallocatePropVals(&props);
. . .
```

Getting and setting session properties

The following sections describe useful tasks you can perform by getting and setting session properties.

Making a document or book active

In addition to finding out which document is active by getting the session's `FP_ActiveDoc` property, you can make a document active by setting this property. For example, the following code makes the document specified by `docId` active:

```
. . .  
F_ObjHandleT docId;  
F_ApiSetId(0, FV_SessionId, FP_ActiveDoc, docId);  
. . .
```

When you make a visible document active, its window gets input focus. On some platforms, the windowing system highlights a window's title bar or brings it to the front.

```
. . .
```

Disabling redisplaying to avoid screen flicker

If you change numerous properties at once, it may cause screen flicker, an effect that occurs when a FrameMaker product executes a long series of changes that aren't user-initiated. By default, FrameMaker products reformat after each change.

You can avoid screen flicker by batching changes. To batch changes, set the `FO_Session` property `FP_Displaying` to `False`. As long as `FP_Displaying` is set to `False`, the FrameMaker product does not refresh the documents in the current session when you or the user changes them. To refresh the documents, you must call `F_ApiRedisplay()` for each changed document.

The syntax for `F_ApiRedisplay()` is:

```
IntT F_ApiRedisplay(F_ObjHandleT docId);
```

For example, to change a number of properties at once, use code similar to the following:

```
. . .  
F_ObjHandleT docId;  
  
F_ApiSetInt(0, FV_SessionId, FP_Displaying, False);  
  
/* Change multiple properties here. */  
  
F_ApiSetInt(0, FV_SessionId, FP_Displaying, True);  
F_ApiRedisplay(docId); /* Must be called for each document */  
. . .
```

While `FP_Displaying` is set to `False`, the FrameMaker product doesn't update the display at all. In some cases, you may want the FrameMaker product to update the display but to delay reformatting documents while you change them. To do this, set the `FO_Session` property `FP_Reformatting` to `False`. After you have reset `FP_Reformatting` to `True`, refresh the documents that you have changed by calling `F_ApiReformat()` for each document.

Getting and setting document properties

The following sections describe useful tasks you can perform by getting and setting document properties.

Getting a document's pathname

A document's absolute pathname is specified by its `FP_Name` property. The following code displays the active document's absolute pathname:

```

. . .
F_ObjHandleT docId;
StringT docName;

/* Get the document ID and name. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
docName = F_ApiGetString(FV_SessionId, docId, FP_Name);
F_ApiAlert(docName, FF_ALERT_CONTINUE_NOTE);

F_Free(docName);
. . .

```

Manipulating document windows

The API provides several properties that allow you to manipulate document and book windows. To change a document window's size and screen location, set the document's `FP_ScreenX`, `FP_ScreenY`, `FP_ScreenWidth`, and `FP_ScreenHeight` properties. To bring the window to the front, set the document's `FP_IsInFront` property.

Setting a document or book title bar

The API allows you to set the title bars of both documents and books. By default, a document or book's title bar displays its name. However, you can make it display another string by setting the document or book's `FP_Label` property to the string. For example, the following code displays the string `MyTitle` in the title bar of the active document:

```

. . .
F_ObjHandleT docId;
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
F_ApiSetString(FV_SessionId, docId, FP_Label, "MyTitle");
. . .

```

After you set a document's title bar, it doesn't change until you reset it or the user closes the document.

Setting the title bar of a document or book doesn't change the name of the document or book file itself. If you set the document or book title bar and the user closes and reopens the document or book, the document or book name appears in the title bar again.

Setting a document or book status bar

The API allows you to set the status bars of both documents and books. If your client conducts extensive processing, it can display status messages in the status bar to inform users of its progress. To set the status bar of a document or a book, set its `FP_StatusLine` property.

The string you set `FP_StatusLine` to remains in the status bar only until a client or the FrameMaker product overwrites it. FrameMaker products overwrite the status bar frequently. For example, every time the user moves the insertion point to a different paragraph in a document, the FrameMaker product redisplay the paragraph format in the status bar.

Enhancing performance by making documents invisible

The API allows you to make a document invisible. Your client can still make changes to an invisible document. If your client needs to batch process multiple documents, using invisible documents can increase its performance considerably.

To make a document invisible, you can use the following code:

```
. . .  
F_ObjHandleT docId;  
F_ApiSetInt(FV_SessionId, docId, FP_IsOnScreen, False);  
. . .
```

You can also open documents invisibly by setting the `FP_MakeVisible` property of the `Open` script to `False`.

.....
IMPORTANT: *Because an invisible document can't get input focus, it can't be the active document specified by the session property `FP_ActiveDoc`. You can't send f-codes to an invisible document.*
.....

Displaying a particular page

Document objects (FO_Doc) have a property named `FP_CurrentPage` that specifies the ID of the *current page*. The current page is the page that appears on the screen. If more than one page appears on the screen, it is the page that appears with a dark border around it. You can make a page current by making the document that contains it the active document and then setting the document's `FP_CurrentPage` property to the page's ID.

For example, the following code displays the second body page and then the first reference page of the active document:

```
. . .
F_ObjHandleT docId, bPg1Id, bPg2Id, rPg1Id;
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);

/* Get second body page ID. */
bPg1Id = F_ApiGetId(FV_SessionId, docId, FP_FirstBodyPageInDoc);
bPg2Id = F_ApiGetId(docId, bPg1Id, FP_PageNext);

if (bPg2Id)
{
    F_ApiSetId(FV_SessionId, docId, FP_CurrentPage, bPg2Id);
    F_ApiAlert("Now at 2nd body page.", FF_ALERT_CONTINUE_NOTE);
}

/* Go to first reference page. */
rPg1Id = F_ApiGetId(FV_SessionId, docId, FP_FirstRefPageInDoc);
if (rPg1Id)
    F_ApiSetId(FV_SessionId, docId, FP_CurrentPage, rPg1Id);
. . .
```

Getting and setting graphic object properties

The following sections describe useful tasks you can perform by getting and setting graphic object properties.

Changing an object's size and location within a frame

Each graphic object has `FP_Height` and `FP_Width` properties, which specify its height (the distance between its highest and lowest points) and its width (the distance between its leftmost and rightmost points). To change an object's size, use `F_ApiSetMetric()` to set these properties. For example, the following code increases a selected object's width by 10 points:

```
. . .
#define pts (MetricT) 65536 /* Frame metric point */
F_ObjHandleT docId, objId;

docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);

/* Get ID of selected object. */
objId = F_ApiGetId(FV_SessionId, docId,
                  FP_FirstSelectedGraphicInDoc);

/* Get object's current width and add 10 points to it. */
if (objId)
    F_ApiSetMetric(docId, objId, FP_Width,
                  F_ApiGetMetric(docId, objId, FP_Width) + 10*pts);
. . .
```

If you set the `FP_Height` and `FP_Width` properties of a polyline or polygon, the API changes all the object's vertices proportionally. If you want to change a polygon or polyline's vertices independently, use `F_ApiSetPoints()` to set its `FP_Points` property. For an example of how to set the `FP_Points` property, see “`F_ApiSetPoints()`” in the FDK Programmer's Reference guide.

All graphic objects have an `FP_LocX` property, which specifies the distance of the object's leftmost point from the left side of the parent frame, and an `FP_LocY` property, which specifies the distance of the object's uppermost point from the top of its parent frame. To change an object's location within a frame, use `F_ApiSetMetric()` to set these properties.

Moving graphics forward or back in the draw order

FrameMaker products maintain the graphic objects in each frame in a linked list. Each graphic object has `FP_PrevGraphicInFrame` and `FP_NextGraphicInFrame` properties that specify the graphic objects before and after it in the list. The order of this list corresponds to the back-to-front draw order. The first object in the list is the first object the FrameMaker product draws, and therefore appears in back of objects later in the list. To move a graphic object forward or back in the draw order, you change its `FP_PrevGraphicInFrame` or `FP_NextGraphicInFrame` property so that it specifies a different object. You need to change only one of these properties. The FrameMaker product automatically changes the other one for you.

It also automatically changes the `FP_PrevGraphicInFrame` or `FP_NextGraphicInFrame` properties of the object's siblings.

To move an object all the way to the back of the objects in a frame, set its `FP_PrevGraphicInFrame` property to 0. To move an object all the way to the front, set its `FP_NextGraphicInFrame` property to 0.

For example, the following code moves a selected graphic object forward one level:

```
. . .
F_ObjHandleT docId, objId, sibId;

docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);

/* Get ID of one (and only one) selected object. */
objId = F_ApiGetId(FV_SessionId, docId,
                  FP_FirstSelectedGraphicInDoc);
if (!objId || F_ApiGetId(docId, objId,
                        FP_NextSelectedGraphicInDoc))
    F_ApiAlert("Select a single object", FF_ALERT_CONTINUE_NOTE);
else
    {
    /* Try to get ID of object in front of selected object. */
    sibId = F_ApiGetId(docId, objId, FP_NextGraphicInFrame);

    /* If there is an object in front, put it behind. */
    if(sibId)
        F_ApiSetId(docId, objId, FP_PrevGraphicInFrame, sibId);
    }
. . .
```


Moving graphic objects to different frames or pages

To move a graphic object to a different frame, set its `FP_FrameParent` property to the ID of that frame. The API automatically changes all the properties that need to be changed to maintain the lists of objects in the object's old and new parent frames.

To move a graphic object to a different page, set its `FP_FrameParent` property to the ID of a frame on that page. All pages have an invisible frame, called a page frame. To put a graphic object directly on a page, set its `FP_FrameParent` property to the ID of the page's page frame. For more information on page frames, see "How the API represents pages" on page 88.

For an example of how to move objects from a frame to a page frame, see "F_ApiSetId()" in the FDK Programmer's Reference guide.

Grouping objects

To group a set of objects, you first use `F_ApiNewGraphicObject()` to create a group (`FO_Group`) object. Then you add the objects to the group object by setting their `FP_GroupParent` properties to the ID of the group object. The objects must be in the same frame as the group object. For information on how to use `F_ApiNewGraphicObject()`, see "Creating graphic objects" on page 365.

To remove an object from a group, set the object's `FP_GroupParent` property to 0.

Copying properties from one graphic object to another

Each type of graphic object has a number of properties, such as `FP_Fill` and `FP_BorderWidth`, which are common to all graphic objects. Some of these properties don't manifest themselves for all graphic objects. For example, rectangles have an `FP_ArrowType` property, although they don't have arrowheads. For a list of properties common to all graphic objects, see "Common graphics properties" in the FDK Programmer's Reference guide.

You can use `F_ApiGetProps()` and `F_ApiSetProps()` to copy common properties from one graphic object to another, as shown in the example in "Getting and setting property lists" on page 293. When you copy properties from one graphic object to another, the objects do not have to be the same type. For example, you can copy the properties from a line to a rectangle. The API copies only the common properties, leaving properties that are specific to the rectangle, such as `FP_RectangleIsSmoothed`, intact.

Getting and setting paragraph properties

The following sections describe useful tasks you can perform by getting and setting paragraph properties.

Applying paragraph and Paragraph Catalog formats

Paragraph (FO_Pgf) objects and Paragraph Catalog format (FO_PgfFmt) objects have the same formatting properties. To apply the properties from a paragraph to a Paragraph Catalog format or from a Paragraph Catalog format to a paragraph, you can use `F_ApiGetProps()` and `F_ApiSetProps()`.

For example, the following code applies the Paragraph Catalog format named `Body` to the paragraph containing the insertion point:

```

. . .
F_PropValsT props;
F_TextRangeT tr;
F_ObjHandleT docId, pgfId, bodyFmtId;
StringT pgfName;

docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);

/* Get ID of the Body Paragraph Catalog format. */
bodyFmtId = F_ApiGetNamedObject(docId, FO_PgfFmt, "Body");
if (!bodyFmtId) return;

/* Get text selection. For more information, see page 323. */
tr = F_ApiGetTextRange(FV_SessionId, docId, FP_TextSelection);
if (tr.beg.objId == 0) return;

/* Get properties from the Body Paragraph Catalog format. */
props = F_ApiGetProps(docId, bodyFmtId);
if (props.len == 0) return;

/* Apply Body properties to paragraph containing insertion
 * point (or the beginning of the text selection).
 */
F_ApiSetProps(docId, tr.beg.objId, &props);
. . .

```

If you have changed the Body format, you may want to reapply it to all paragraphs that are tagged Body. To change these paragraphs, you must traverse every paragraph in the document, determine if it's tagged Body, and set its properties if it is. You can do this by adding the following code to the code shown above:¹

```
. . .
pgfId = F_ApiGetId(FV_SessionId, docId, FP_FirstPgfInDoc);

while (pgfId)
{
    /* Get each paragraph's tag and see if it's Body. */
    pgfName = F_ApiGetString(docId, pgfId, FP_Name);
    if (F_StrEqual((StringT)"Body", pgfName))
        F_ApiSetProps(docId, pgfId, &props);
    F_Free(pgfName);
    pgfId = F_ApiGetId(docId, pgfId, FP_NextPgfInDoc);
}
. . .
```

Adding tabs

To get and set the tabs for a paragraph or Paragraph Catalog format, use `F_ApiGetTabs()` and `F_ApiSetTabs()`.

The syntax for `F_ApiGetTabs()` and `F_ApiSetTabs()` is:

```
F_TabsT F_ApiGetTabs(F_ObjHandleT docId,
    F_ObjHandleT objId,
    IntT propNum);
```

```
VoidT F_ApiSetTabs(F_ObjHandleT docId,
    F_ObjHandleT objId,
    IntT propNum,
    F_TabsT *setVal);
```

.....
1. Some examples in this chapter use FDE functions, such as `F_StrEqual()`, `F_Alloc()`, and `F_Realloc()`. For more information on using the FDE and these functions, see Part III, "Frame Development Environment (FDE)."

This argument	Means
docId	The ID of the document containing the paragraph or paragraph format whose tabs you want to query or set.
objId	The ID of the paragraph or paragraph format whose tabs you want to query or set.
propNum	The property to query. Specify FP_Tabs.
setVal	The F_TabsT structure to which to set the property.

The F_TabsT structure is defined as:

```
typedef struct {
    UIntT len; /* The number of tabs in val */
    F_TabT *val; /* Structures that describe the tabs */
} F_TabsT;
```

The F_TabT structure is defined as:

```
typedef struct {
    MetricT x; /* Offset from paragraph's left margin */
    UCharT type; /* Constant for tab type, e.g. FV_TAB_RIGHT */
    StringT leader; /* Characters before tab, e.g. "." */
    UCharT decimal; /* Character for decimal tab, e.g. "." */
} F_TabT;
```

When you get the tabs for a paragraph or paragraph format, the API returns them in left-to-right order in the `val` array. However, when you insert a tab, you *don't* have to insert it in this order. You just add it to the end of the `val` array. When you call

`F_ApiSetTabs()`, the API sorts the tabs for you. For example, the following code adds a 4-inch decimal tab to the Body paragraph format:

```
. . .
#include "fmemory.h"
#define in (MetricT) (65536 * 72)
F_ObjHandleT docId, pgfFmtId;
F_TabsT tabs;

/* Get the ID for the Body paragraph format. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
pgfFmtId = F_ApiGetNamedObject(docId, FO_PgfFmt, "Body");
if (!pgfFmtId) return;

/* Get the tabs and allocate space for new tab. */
tabs = F_ApiGetTabs(docId, pgfFmtId, FP_Tabs);
if (tabs.len++)
    tabs.val=(F_TabT*)F_Realloc(tabs.val,
        tabs.len*sizeof(F_TabT), NO_DSE);
else
    tabs.val = (F_TabT*) F_Alloc(sizeof(F_TabT),NO_DSE);

/* Add the tab. */
tabs.val[tabs.len-1].type = FV_TAB_DECIMAL;
tabs.val[tabs.len-1].x = 4*in;
tabs.val[tabs.len-1].decimal = ',';
tabs.val[tabs.len-1].leader = F_StrCopyString(" ");

/* Set paragraph format's tabs property to the array of tabs. */
F_ApiSetTabs(docId, pgfFmtId, FP_Tabs, &tabs);
F_ApiDeallocateTabs(&tabs);
. . .
```

Getting and setting book properties

To rearrange book components, you change their `FP_PrevComponentInBook` and `FP_NextComponentInBook` properties.

For example, to move the first component in a book down one position, you can use the following code:

```
. . .  
F_ObjHandleT bookId, firstC, nextC;  
  
bookId = F_ApiGetId(0, FV_SessionId, FP_ActiveBook);  
firstC = F_ApiGetId(FV_SessionId, bookId,  
                   FP_FirstComponentInBook);  
nextC = F_ApiGetId(bookId, firstC, FP_NextComponentInBook);  
  
if (nextC)  
    F_ApiSetId(bookId, firstC, FP_PrevComponentInBook, nextC);  
else  
    F_ApiAlert("Only one component.", FF_ALERT_CONTINUE_NOTE);  
. . .
```

Getting and setting FrameMaker properties

There are some special issues involved in getting and setting properties in structured FrameMaker documents. The following sections discuss some of these issues.

Traversing elements

To traverse the elements in a structured document, you use slightly different code than you would use to traverse other objects, such as paragraphs. If you want to traverse all the elements in a document, you can't query only `FP_NextSiblingElement` properties. You must also recursively traverse each element's child elements. For example, the following function prints the IDs of all the elements in a specified element:

```
. . .  
VoidT traverseElement(F_ObjHandleT docId,  
                     F_ObjHandleT elementId)  
  
{  
    StringT name;  
  
    if (elementId)  
    {  
        elementId = F_ApiGetId(docId, elementId,  
                               FP_FirstChildElement);  
        while(elementId)  
        {  
            F_Printf(NULL, (StringT) "Element ID is 0x%x.\n",  
                     elementId);  
            traverseElement(docId, elementId);  
            elementId = F_ApiGetId(docId, elementId,  
                                   FP_NextSiblingElement);  
        }  
    }  
}  
. . .
```

Manipulating format change list properties


Most object types in the FDK have a single list of properties that applies to all objects of that type. For example, if you call `F_ApiGetProps()` for any `FO_Pgf` object in a document, it will always return the same list of properties. The values of the properties may be different for each paragraph, but the list of properties will always be the same. This is not the case with `FO_FmtChangeList` objects.

All `FO_FmtChangeList` objects have the following common properties:

- FP_Name
- FP_NextFmtChangeListInDoc
- FP_PgfCatalogReference

However, individual `FO_FmtChangeList` objects can have different sets of additional properties, depending on what formatting characteristics they set. An `FO_FmtChangeList` object can have all the properties listed under “Format change lists” in the FDK Programmer’s Reference, or it may have just a small subset of these properties.

For example, the format change list in Figure 5-1 has only the common properties listed above and the `FP_FontFamily` property. If you call `F_ApiGetProps()` for this format change list, the function returns only four properties: the three common properties listed above and the `FP_FontFamily` property.



Format change list: Code
Default font properties
Family: Courier

Figure 5-1 *Format change list*

If you attempt to use an `F_ApiGetPropertyType()` function to get a property that a format change list doesn’t have, the function fails, setting `FA_errno` to `FE_PropNotSet`.

Adding properties to a format change list

To add a property to a format change list, you just set the property on the `FO_FmtChangeList` object. You can do this by calling an `F_ApiSetPropertyType()` function or by creating a property list containing the property and calling `F_ApiSetProps()` to set the list on the object. For example, the following code uses these two methods to add properties to the Code format change list:

```
. . .
#define pts (MetricT) 65536
F_PropValsT props;
F_ObjHandleT docId, changeListId;

docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);

/* Create the Code change list. */
changeListId = F_ApiNewNamedObject(docId, FO_FmtChangeList,
                                   "Code");

/* Add the FP_PairKern property to turn pair kerning off. */
F_ApiSetInt(docId, changeListId, FP_PairKern, False);

/* Set up list with FP_FontSize property to set size to 10. */
props = F_ApiAllocatePropVals(1);
props.val[0].propIdent.num = FP_FontSize;
props.val[0].propVal.valType = FT_Metric;
props.val[0].propVal.u.ival = 10*pts;

F_ApiSetProps(docId, changeListId, &props);
. . .
```

Note that the `F_ApiSetProps()` call in the code above only adds the `FP_FontSize` property. It does not affect the other properties of the format change list.

Removing properties from a format change list

To remove a property from a format change list, call `F_ApiDeletePropByName()`. For example, the following code removes the `FP_PairKern` property from the Code format change list:

```

. . .
F_ObjHandleT docId, changeListId;

docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
changeListId = F_ApiGetNamedObject(docId, FO_FmtChangeList,
                                   "Code");
if (!changeListId) return;
F_ApiDeletePropByName(docId, changeListId, "FP_PairKern");
. . .

```

Setting format rule clause properties

Format rule clauses (`FO_FmtRuleClause` objects) have several properties that are only indirectly settable. For example, you can't set the `FP_RuleClauseType` property directly. You can only set it indirectly by setting one of the following properties:

- `FP_FormatName`
- `FP_FmtChangeListTag`
- `FP_FmtChangeList`
- `FP_SubFmtRule`

For example, if you set `FP_FmtChangeListTag` to `Code`, FrameMaker automatically sets `FP_RuleClauseType` to `FV_RC_CHANGELIST_TAG`.

You can also set `FP_FmtChangeList` and `FP_SubFmtRule` only indirectly—by calling `F_ApiSubObject()` to add a change list or subformat rule to the format rule clause or by calling `F_ApiDelete()` to delete a change list or subformat rule. For an example of how to use `F_ApiSubObject()`, see “Creating format rules, format rule clauses, and format change lists” on page 373.

Setting element definition properties

Element definitions have the following properties that are only indirectly settable:

- `FP_TextFmtRules`
- `FP_ObjectFmtRules`
- `FP_PrefixRules`

- FP_SuffixRules
- FP_FirstPgfRules
- FP_LastPgfRules

Each of these properties specifies a list of format rules. You can't directly add a format rule to one of these lists. Instead, you must call `F_ApiSubObject()`.

`F_ApiSubObject()` creates an `FO_FmtRule` object and adds it to the end of the specified list.

You also can't directly remove a format rule from a list. Instead, you must call `F_ApiDelete()` to delete the format rule. For example, the following code adds and deletes a text format rule:

```
. . .  
F_ObjHandleT docId, paraEdefId, fmtRuleId;  
  
paraEdefId = F_ApiGetNamedObject(docId, FO_ElementDef, "Para");  
  
/* Add a text format rule to the element definition. */  
fmtRuleId = F_ApiSubObject(docId, paraEdefId,  
                           FP_TextFmtRules);  
  
/* Delete the text format rule. */  
F_ApiDelete(docId, fmtRuleId);  
. . .
```

Determining the formatting that applies to an element

To determine the formatting that applies to an element, you first get the following properties:

- FP_MatchingTextClauses
- FP_MatchingObjectClauses
- FP_MatchingPrefixClauses
- FP_MatchingSuffixClauses
- FP_MatchingFirstPgfClauses
- FP_MatchingLastPgfClauses

Each of these properties specifies a list of format rule clauses that applies to the element. Getting these properties only for the element itself is not sufficient to determine the element's formatting. These properties specify only the format rule clauses that are in the element definition's format rules (that is, the format rules specified by the element definition's `FP_TextFmtRules`, `FP_ObjectFmtRules`, `FP_PrefixRules`,

`FP_SuffixRules`, `FP_FirstPgfRules`, and `FP_LastPgfRules` properties). In order to fully determine the element's formatting, you must find any applicable format rule clauses that the element inherits from its ancestor elements. To determine whether an element inherits format rule clauses from ancestor elements, you must traverse up the structure tree and check the `FP_MatchingClauses` properties for each ancestor element.

Determining which element contains an object

Frequently, it is useful to determine which element contains an object, such as a cross-reference, a marker, or a table. The following table lists the properties you query to get the ID of an object's containing element.

Object	Property that returns ID of containing element
<code>FO_Marker</code>	<code>FP_Element</code>
<code>FO_Fn</code>	
<code>FO_XRef</code>	
<code>FO_Var</code>	
<code>FO_AFrame</code>	
<code>FO_Row</code>	
<code>FO_Cell</code>	
<code>FO_Tbl^a</code>	<code>FP_TblElement</code>
	<code>FP_TblTitleElement</code>
	<code>FP_TblHeaderElement</code>
	<code>FP_TblBodyElement</code>
	<code>FP_TblFooterElement</code>

a. If a table has a title or different types of rows, it can comprise several elements.

To determine the ID of the object an element contains, you query the element's `FP_Object` property.

Specifying client data for an element

The `FO_Element` property `FP_UserString` allows your client to store its own data with individual structural elements. The `FP_UserString` property is persistent between sessions; after a client sets it, it remains the same until a client resets it. If an element is cut and pasted, it retains its `FP_UserString` property. If an element is copied and pasted, both the original element and the pasted element retain the `FP_UserString` property.

Improving performance in FrameMaker clients

If you are using the API to create structured documents, you may need to add a large number of elements or element definitions at a time. By default, FrameMaker validates elements and applies format rules each time you add an element or element definition. This can decrease performance considerably. To keep FrameMaker from validating elements and applying format rules, set the `FO_Session` properties `FP_Validating` and `FP_ApplyFmtRules` to `False`.

Manipulating Text

.....

.....

This chapter describes how to use the API to manipulate text in Frame documents. Specifically, it discusses how to:

- Retrieve text from a document
- Get and set the location of the insertion point or current text selection
- Add and delete text
- Get and set text formatting
- Programmatically execute Clipboard operations

To better understand the material in this chapter, you may want to learn more about how the API represents text. For information on this subject, see “Text” on page 114.

Getting text

Text in Frame documents is contained in objects, such as `FO_Cell`, `FO_Element`, `FO_Fn`, `FO_Pgf`, `FO_TextLine`, `FO_Var`, `FO_SubCol`, `FO_TextFrame`, and `FO_Flow` objects. To get text, you must get the ID of the object that contains it. For information on getting object IDs, see “Getting the IDs of the objects you want to change” on page 280.

Once you have the ID of an object that contains text, you use `F_ApiGetText()` to retrieve the text.

The syntax for `F_ApiGetText()` is:

```
F_TextItemsT F_ApiGetText(F_ObjHandleT docId,
    F_ObjHandleT objId,
    IntT flags);
```

This argument	Means
<code>docId</code>	The ID of the document containing the object for which you want to get text.

This argument	Means
objId	The ID of the object (FO_Flow, FO_Element, FO_Fn, FO_Pgf, FO_Cell, FO_SubCol, FO_TextFrame, FO_TextLine, or FO_Var) containing the text.
flags	Bit flags that specify the type of text items to retrieve. To get specific types of text items, OR the constants that represent them (for example, FTI_FlowBegin and FTI_String) into flags. To get all types of text items, specify -1. For a complete list of the constants that represent text item types, see “F_ApiGetText()” in the FDK Programmer’s Reference guide.

The `F_TextItemsT` structure contains an array of *text items*. Each string of characters with common character and condition properties, each anchor, and each line or column break in the text constitutes a separate text item.

`F_TextItemsT` is defined as:

```
typedef struct {
    UIntT len; /* The number of text items */
    F_TextItemT *val; /* Array of text items */
} F_TextItemsT;
```

The API represents each text item with an `F_TextItemT` structure. `F_TextItemT` is defined as:

```
typedef struct {
    IntT offset; /* Characters from beginning */
    IntT dataType; /* Text item type, e.g. FTI_String */
    union {
        StringT sdata; /* String if the type is FTI_String */
        F_ObjHandleT idata; /* ID if item is an anchor */
    } u;
} F_TextItemT;
```


If a text item represents a string of characters, `F_TextItemT.dataType` is set to `FTI_String` and `F_TextItemT.u.sdata` contains the string. If the text item represents an anchor, `F_TextItemT.dataType` is set to a constant indicating the anchor type (for example, `FTI_TblAnchor`) and `F_TextItemT.u.idata` contains the ID of the anchored object (for example, an `FO_Tbl` object).

For more information on the `F_TextItemsT` structure, see “How the API represents text” on page 114.

After you are finished with an `F_TextItemsT` structure, free the memory that it uses with `F_ApiDeallocateTextItems()`. The syntax for `F_ApiDeallocateTextItems()` is:

```
VoidT F_ApiDeallocateTextItems(F_TextItemsT *itemsp);
```

where `itemsp` is the `F_TextItemsT` structure that you want to free.

If you call `F_ApiGetText()` for a structural element (`FO_Element` object), the returned information depends on the type of element, as shown in the following table:

Element's <code>FP_ElementType</code> value	Information returned by <code>F_ApiGetText()</code>
<code>FV_FO_CONTAINER</code>	All the text items from the beginning to the end of the element.
<code>FV_FO_SYS_VAR</code>	All the text items from the beginning to the end of the variable.
<code>FV_FO_XREF</code>	All the text items from the beginning to the end of the cross-reference.
<code>FV_FO_FOOTNOTE</code>	All the text items from the beginning to the end of the footnote.
<code>FV_FO_TBL_TITLE</code>	All the text items from the beginning to the end of the table title.
<code>FV_FO_TBL_CELL</code>	All the text items from the beginning to the end of the cell.

Element's FP_ElementType value	Information returned by F_ApiGetText()
FV_FO_TBL_HEADING	Nothing. F_ApiGetText() fails.
FV_FO_TBL_BODY	
FV_FO_TBL_FOOTING	
FV_FO_MARKER	
FV_FO_TBL	
FV_FO_GRAPHIC	
FV_FO_EQN	
FV_FO_TBL_ROW	

Example

The following code retrieves and prints the text in the active document's main flow to the console. It retrieves and prints only strings and line ends.

```

. . .
#include "futils.h"
F_ObjHandleT docId, flowId;
IntT i;
F_TextItemsT tis;
F_TextItemT *ip;

/* Get IDs for active document and main flow. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
flowId = F_ApiGetId(FV_SessionId, docId, FP_MainFlowInDoc);

tis = F_ApiGetText(docId, flowId, FTI_String | FTI_LineEnd);

/* Traverse text items and print strings and line ends. */
for (i=0; i<tis.len; i++)
{
    ip = &tis.val[i];
    if (ip->dataType == FTI_String)
        F_Printf(NULL, "%s", ip->u.sdata);
    else F_Printf(NULL, "\n");
}
F_ApiDeallocateTextItems(&tis);
. . .

```

Getting and setting the insertion point or text selection

The Frame API uses the document property `FP_TextSelection` to specify the insertion point or text selection in a document. This property specifies a text range, or `F_TextRangeT` structure, which is defined as:

```
typedef struct {
    F_TextLocT beg; /* Beginning of the text range */
    F_TextLocT end; /* End of the text range */
} F_TextRangeT;
```

The `F_TextLocT` structure, which specifies a *text location* (a particular point in text), is defined as:

```
typedef struct{
    F_ObjHandleT objId; /* Object that contains the text */
    IntT offset; /* Characters from beginning */
} F_TextLocT;
```

If a range of text is selected, `FP_TextSelection` specifies a selection; `F_TextRangeT.beg` and `F_TextRangeT.end` specify the beginning and end of the selection. If there is an insertion point, `FP_TextSelection` specifies an insertion point; `F_TextRangeT.beg` and `F_TextRangeT.end` are the same—both specify the location of the insertion point. If there is no text selection or insertion point, the `objId` and `offset` fields of both `F_TextRangeT.beg` and `F_TextRangeT.end` are set to 0.

For example, suppose the first five characters of the first paragraph on the page shown in Figure 6-1 are selected.

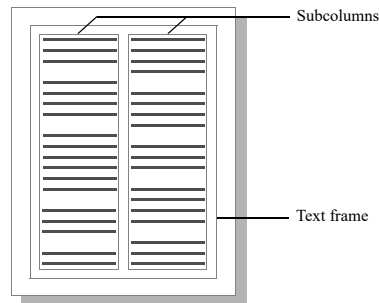


Figure 6-1 Page with text frame containing two subcolumns

The fields of the `F_TextRangeT` structure specified by the document's `FP_TextSelection` property have the following values:

Field	Value
<code>beg.objId</code>	ID of the first paragraph
<code>beg.offset</code>	0
<code>end.objId</code>	ID of the first paragraph
<code>end.offset</code>	5

If no text is selected and the insertion point is at the very beginning of the paragraph, the fields of the `F_TextRangeT` structure have the following values:

Field	Value
<code>beg.objId</code>	ID of the first paragraph
<code>beg.offset</code>	0
<code>end.objId</code>	ID of the first paragraph
<code>end.offset</code>	0

.....
IMPORTANT: *A valid text range can span multiple paragraphs, subcolumns, or text frames. It can't span multiple flows, footnotes, table cells, or text lines.*

It is possible for a document to have no text selection or insertion point at all. This can occur in the following circumstances:

- One or more graphic objects in the document are selected
- One or more entire table cells in the document are selected
- There is no selection of any type in the document

If a document has no text selection or insertion point at all, the fields of the `F_TextRangeT` structure have the following values:

Field	Value
<code>beg.objId</code>	0
<code>beg.offset</code>	0

Field	Value
<code>end.objId</code>	0
<code>end.offset</code>	0

For more information on the different types of selection and the properties that represent it, see “How the API represents the selection in a document” on page 82.

To get and set a document’s insertion point (or text selection), use `F_ApiGetTextRange()` and `F_ApiSetTextRange()` to get and set its `FP_TextSelection` property.

The syntax for these functions is:

```
F_TextRangeT F_ApiGetTextRange(F_ObjHandleT parentId,
    F_ObjHandleT objId,
    IntT propNum);
```

```
VoidT F_ApiSetTextRange(F_ObjHandleT parentId,
    F_ObjHandleT objId,
    IntT propNum,
    F_TextRangeT *setVal);
```

This argument	Means
<code>parentId</code>	The ID of the object containing <code>objId</code> . If <code>objId</code> specifies a document ID, <code>parentId</code> should specify <code>FV_SessionId</code> . If <code>objId</code> specifies a flow, text frame, or table cell ID, <code>parentId</code> should specify the ID of the document that contains it.
<code>objId</code>	The ID of the object whose property you want to get or set. To get or set the insertion point or text selection in a document, specify the document’s ID.
<code>propNum</code>	The property to get or set. To get or set the insertion point or text selection in a document, set <code>propNum</code> to <code>FP_TextSelection</code> .
<code>setVal</code>	The text range to which to set the property.

The `beg.objId` and `end.objId` fields of the `F_TextRangeT` structure returned by `F_ApiGetTextRange()` always specify paragraph or text line IDs. The `beg.objId` and `end.objId` fields of the `F_TextRangeT` structure that you pass to `F_ApiSetTextRange()` can specify paragraph or text line IDs, but they can also specify flow, footnote, subcolumn, table cell, or text frame IDs.

For example, to set the insertion point at the beginning of the first paragraph on the page shown in Figure 6-1, you can use the following code:

```
. . .
F_ObjHandleT docId, pgfId;
F_TextRangeT tr;
. . .
/* Get document and paragraph IDs here. */
. . .
/* Create text range that specifies an insertion point. */
tr.beg.objId = tr.end.objId = pgfId;
tr.beg.offset = tr.end.offset = 0;

/* Set document's insertion point. */
F_ApiSetTextRange(FV_SessionId, docId, FP_TextSelection, &tr);
. . .
```

Instead of setting `tr.beg.objId` and `tr.end.objId` to the ID of the first paragraph, you can set them to the ID of the A flow, the text frame, or the left subcolumn. For example, the following code also sets the insertion point at the beginning of the first paragraph on the page shown in Figure 6-1:

```
. . .
F_ObjHandleT docId, flowId;
F_TextRangeT tr;
. . .
/* Get document and flow IDs here. */
. . .
tr.beg.objId = tr.end.objId = flowId;
tr.beg.offset = tr.end.offset = 0;

/* Set document's insertion point. */
F_ApiSetTextRange(FV_SessionId, docId, FP_TextSelection, &tr);
. . .
```

The `beg.offset` and `end.offset` fields of the `F_TextRangeT` structure returned by `F_ApiGetTextRange()` always specify offsets relative to the beginning of a paragraph or text line object. The `beg.offset` and `end.offset` fields of the `F_TextRangeT` structure that you pass to `F_ApiSetTextRange()` can specify offsets relative to the beginning of an object, but they can also use the special value `FV_OBJ_END_OFFSET`. `FV_OBJ_END_OFFSET` specifies the offset of the last character in the object containing the text range. To specify offsets near the end of an object, you can add or subtract integers from `FV_OBJ_END_OFFSET`. For example, the following code selects the last five characters in a paragraph and the end of paragraph symbol:

```
. . .
F_ObjHandleT docId, pgfId;
F_TextRangeT tr;
. . .
/* Get document and paragraph IDs here. */
. . .
tr.beg.objId = tr.end.objId = pgfId;
tr.beg.offset = FV_OBJ_END_OFFSET - 6;
tr.end.offset = FV_OBJ_END_OFFSET;
F_ApiSetTextRange(FV_SessionId, docId, FP_TextSelection, &tr);
. . .
```

The following code selects all the text in a cell:

```
. . .
F_ObjHandleT docId, cellId;
F_TextRangeT tr;
. . .
/* Get document and cell IDs here. */
. . .
tr.beg.objId = tr.end.objId = cellId;
tr.beg.offset = 0;
tr.end.offset = FV_OBJ_END_OFFSET;
F_ApiSetTextRange(FV_SessionId, docId, FP_TextSelection, &tr);
. . .
```

Getting the text in a text range

To get the text in a specific text range, use `F_ApiGetTextForRange()`. The syntax for `F_ApiGetTextForRange()` is:

```
F_TextItemsT F_ApiGetTextForRange(F_ObjHandleT docId,
    F_TextRangeT *tr,
    IntT flags);
```

This argument	Means
<code>docId</code>	The ID of the document containing the text range.
<code>tr</code>	The text range containing the text you want to get.
<code>flags</code>	Bit flags that specify the type of text items to retrieve. For a complete list of the constants that represent text item types, see “ <code>F_ApiGetText()</code> ” in the FDK Programmer’s Reference guide.

For example, the following code gets the selected text in the active document:

```
. . .
F_ObjHandleT docId;
F_TextRangeT tr;
F_TextItemsT tis;

docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tr = F_ApiGetTextRange(FV_SessionId, docId, FP_TextSelection);

/* If there’s just an insertion point, no text is selected. */
if(tr.beg.objId == tr.end.objId
    && tr.beg.offset == tr.end.offset) return;

tis = F_ApiGetTextForRange(docId, &tr, FTI_String);
. . .
```


Getting and setting table selections

If a table contains cells that are selected, you can get the table's ID by querying the document property, `FP_SelectedTbl`. For more information, see “Getting the IDs of selected tables and table rows” on page 287.

If a range of text that includes several tables is selected, you can get the tables' IDs by calling `F_ApiGetText()` and retrieving the `FTI_TblAnchor` text items for the selection. Each `FTI_TblAnchor` text item specifies the ID of a table.

To make the selection in a document include several tables, set the text selection so that it includes the text that contains the tables' anchors. To make the selection include specific rows and columns within a single table, call `F_ApiMakeTblSelection()`. For more information, see “`F_ApiMakeTblSelection()`” on page 288 in the FDK Programmer's Referenceguide.

Element ranges in structured tables

If the current element range is within a cell, or if it indicates a selected table part, you can get the table's ID by querying the document property, `FP_SelectedTbl`. However, it's possible for a client to set the current element range to a point between table part elements. In this case, the document property, `FP_SelectedTbl` is `NULL`. For this reason, you cannot always use `FP_SelectedTbl` to determine whether the current element range is in a table.

If `FP_SelectedTbl` returns `NULL`, the following code determines whether the current element location is within a table, as well as the type of the parent element:

```

. . .
F_ObjHandleT docId;
F_PropValT propVal;
F_ElementRangeT er;

. . .
/* Get the ID of the active document. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
if(!docId) return;

er = F_ApiGetElementRange(FV_SessionId, docId,
FP_ElementSelection);

/* First check to see if there is a selected table. */
propVal = F_ApiGetPropVal(FV_SessionId, docId, FP_SelectedTbl);
if (!propVal.propVal.u.ival) {

/* Now determine whether the current element range is within a
table. */
    propVal = F_ApiGetPropVal(docId, er.beg.parentId,
FP_ElementType);
    if(propVal.propVal.u.ival == FV_FO_TBL) {
        F_Printf(NULL, (StringT)"You are in a table\n");
    } else if(propVal.propVal.u.ival == FV_FO_TBL_TITLE) {
        F_Printf(NULL, (StringT)"You are in a table title\n");
    } else if(propVal.propVal.u.ival == FV_FO_TBL_HEADING) {
        F_Printf(NULL, (StringT)"You are in a table heading\n");
    } else if(propVal.propVal.u.ival == FV_FO_TBL_BODY) {
        F_Printf(NULL, (StringT)"You are in a table body\n");
    } else if(propVal.propVal.u.ival == FV_FO_TBL_FOOTING) {
        F_Printf(NULL, (StringT)"You are in a table footing\n");
    } else if(propVal.propVal.u.ival == FV_FO_TBL_ROW) {
        F_Printf(NULL, (StringT)"You are in a table row\n");
    } else {

```

```

        F_Printf(NULL, (StringT) "You are not in a table at
all\n");
    }
}

/* Be sure to deallocate memory for the property value. */
F_ApiDeallocatePropVal(&propVal);

```

Getting and setting the structural element selection

Although you can get and set selected structural elements in a FrameMaker document by getting and setting the text selection, it is usually easier to use the following functions:

- `F_ApiGetElementRange()` gets the structural element selection in a document or book.
- `F_ApiSetElementRange()` sets the structural element selection in a document or book.

The syntax for these functions is:

```

F_ElementRangeT F_ApiGetElementRange(
    F_ObjHandleT docId,
    F_ObjHandleT objId,
    IntT propNum);

VoidT F_ApiSetElementRange(
    F_ObjHandleT docId,
    F_ObjHandleT objId,
    IntT propNum,
    F_ElementRangeT *setVal);

```

This argument	Means
docId	The object containing objId. To get or set the element selection in a document, specify FV_SessionId.
objId	The ID of the document or book in which you want to get or set the element selection.
propNum	The property to get or set. To get or set the element selection, specify FP_ElementSelection.
setVal	The element range to set the property to.

The `F_ElementRangeT` structure is defined as:

```
typedef struct {
    F_ElementLocT beg; /* Beginning of the element range. */
    F_ElementLocT end; /* End of the element range. */
} F_ElementRangeT;
```

The `F_ElementLocT` structure specifies a location within an element. It is defined as:

```
typedef struct {
    F_ObjHandleT parentId; /* Parent element ID. */
    F_ObjHandleT childId; /* Child element ID. */
    IntT offset; /* Offset within child/parent element. */
} F_ElementLocT;
```

For information on how `FrameMaker` sets the fields of the `F_ElementRangeT` structure specified by `FP_ElementSelection` to represent different types of selection, see “How the API represents the element selection in a structured `FrameMaker` document” on page 83. For examples of how to get and set element selections, see “`F_ApiGetElementRange()`” and “`F_ApiSetElementRange()`” in the `FDK Programmer’s Reference guide`.

To traverse the selection returned by `F_ApiGetElementRange()`, traverse the child elements of the element specified by `beg.childId`. Then traverse its sibling elements and all of their child elements until you reach the element specified by `end.childId`. To traverse an element’s child elements, you query its `FP_FirstChildElement` property and then query each child element’s `FP_NextSiblingElement` property. To traverse an element’s siblings, you query its `FP_NextSiblingElement` property and then query each sibling element’s `FP_NextSiblingElement` property.

Adding and deleting text

To add and delete text, use `F_ApiAddText()` and `F_ApiDeleteText()`.

The syntax for these functions is:

```
F_TextLocT F_ApiAddText(F_ObjHandleT docId,
    F_TextLocT *textLocp,
    StringT text);
```

This argument	Means
<code>docId</code>	The ID of the document to which you'll add text
<code>textLocp</code>	The point in text (text location) at which you'll add text
<code>text</code>	The text to add

```
IntT F_ApiDeleteText(F_ObjHandleT docId,
    F_TextRangeT *textRangep);
```

This argument	Means
<code>docId</code>	The ID of the document to delete text from
<code>textRangep</code>	The text range to delete

`F_ApiAddText()` returns the text location at the end of the text that was added.

`F_ApiDeleteText()` deletes any objects, such as tables and markers, anchored in the text it deletes.

To add text to, or delete text from, a text inset, you must first unlock it by setting its `FP_TiLocked` property to `False`. After you are done adding or deleting text, relock the inset by setting its `FP_TiLocked` property to `True`.

To specify special characters, line breaks, or paragraph breaks when you add text, use octal codes within the text string. For example, to specify an em dash, use `\321`. For more information on special characters, see “How the API represents special characters” on page 120. For a list of the characters in the FrameMaker product character set and the corresponding codes, see “Character Sets” in your FrameMaker product user documentation.

Example

The following code adds some text at the insertion point (or the beginning of the current text selection) and then deletes it. The text has a dagger (†) at the end of it.

```
. . .
F_TextLocT trm;
F_TextRangeT tr;
F_ObjHandleT docId;

/* Get current text selection. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tr = F_ApiGetTextRange(FV_SessionId, docId,
                       FP_TextSelection);

/* Return if there is no selection or IP. */
if(!tr.beg.objId) return;

/* Insert text at insertion point or beginning of selection.
 * Use the octal code 240 to display the dagger.
 */
trm = F_ApiAddText(docId, &tr.beg, "Here's some text.\240");

F_ApiAlert("Now we'll delete it.", FF_ALERT_CONTINUE_NOTE);

/* Set tr to end at end of the added text. Then delete it. */
tr.end.offset = trm.offset;
F_ApiDeleteText(docId, &tr);

. . .
```

Adding text to table cells

To add text to a table cell, you must first get the ID of the cell. To do this, you traverse from the table to the row containing the cell, and then to the cell. Once you have the ID of the cell that you want to add text to, you add text to it by calling `F_ApiAddText()`.

The following code adds some text to the first cell in the first row of the selected table in the active document:

```
. . .  
F_TextLocT ip;  
F_ObjHandleT docId, tblId, rowId, cellId;  
  
/* Get the document and selected table IDs. */  
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);  
tblId = F_ApiGetId(FV_SessionId, docId, FP_SelectedTbl);  
  
/* Get the ID of the first row and cell. */  
rowId = F_ApiGetId(docId, tblId, FP_FirstRowInTbl);  
cellId = F_ApiGetId(docId, rowId, FP_FirstCellInRow);  
  
/* Set up text location at beginning of cell. */  
ip.objId = cellId;  
ip.offset = 0;  
  
F_ApiAddText(docId, &ip, "This text appears in the cell.");  
. . .
```

For an example of how to create a table and add text to its title, see “Creating tables” on page 376.

Getting and setting text formatting

Although the API doesn't represent text as objects, the characters in text have properties. Each character has a property list describing its font, color, condition tags, and other character formatting attributes. The API provides special functions to get and set the properties in this list.

You can also get and set text formatting by getting and setting paragraph, paragraph format, and character format properties. For more information, see "Getting and setting paragraph properties" on page 306.

Getting text properties

To get an individual property for a character, use `F_ApiGetTextPropVal()`. To get the entire list of text properties for a character, use `F_ApiGetTextProps()`.

The syntax for these functions is:

```
F_PropValT F_ApiGetTextPropVal(F_ObjHandleT docId,
    F_TextLocT *textLocp,
    IntT propNum);
```

```
F_PropValsT F_ApiGetTextProps(F_ObjHandleT docId,
    F_TextLocT *textLocp);
```

This argument	Means
<code>docId</code>	The ID of the document containing the character.
<code>textLocp</code>	The text location of the character that you want to get text properties for. The returned properties are the properties that apply to the character to the right of the specified location.
<code>propNum</code>	The text property, such as <code>FP_FontFamily</code> or <code>FP_FontSize</code> , that you want to get.

The API allocates the returned properties. Use `F_ApiDeallocatePropVal()` or `F_ApiDeallocatePropVals()` to free the properties when you're done with them.

You can get the text properties for only one character at a time, because they can be different for each character. For more information on how the API represents characters and text properties, see "How the API represents text" on page 114.

Example

The following code gets the name of the character tag for the character to the right of the insertion point:

```

. . .
F_TextRangeT tr;
F_PropValT prop;
F_ObjHandleT docId;

/* Get the current insertion point. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tr = F_ApiGetTextRange(FV_SessionId, docId, FP_TextSelection);
if(!tr.beg.objId) return;

prop = F_ApiGetTextPropVal(docId, &tr.end, FP_CharTag);

F_Printf(NULL, "The character tag is %s.\n",
         prop.propVal.u.sval);
. . .

```

Setting text properties

To set the text properties for a text range, use `F_ApiSetTextPropVal()` or `F_ApiSetTextProps()`.

The syntax for these functions is:

```

VoidT F_ApiSetTextPropVal(F_ObjHandleT docId,
    F_TextRangeT *textRangep,
    F_PropValT *setVal);

VoidT F_ApiSetTextProps(F_ObjHandleT docId,
    F_TextRangeT *textRangep,
    F_PropValsT *setVal);

```

This argument	Means
<code>docId</code>	The ID of the document containing the text
<code>textRangep</code>	The text range
<code>setVal</code>	The property or property list to apply to the text range

Applying a character format to text

To apply a character format to a text range, copy the property list of the `FO_CharFmt` object that represents the character format to the text range. For example, to apply the character format named `Emphasis` to the current text selection, use the following code:

```

. . .
F_TextRangeT tr;
F_PropValsT props;
F_ObjHandleT docId, charFmtId;
IntT i;

docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
if (!docId) return;

/* Get current text selection. Return if there isn't one.*/
tr = F_ApiGetTextRange(FV_SessionId, docId, FP_TextSelection);
if (!tr.beg.objId) return;

/* Get Emphasis properties. */
charFmtId = F_ApiGetNamedObject(docId, FO_CharFmt, "Emphasis");

props = F_ApiGetProps(docId, charFmtId);

/* Apply properties to selection. */
F_ApiSetTextProps(docId, &tr, &props);
. . .

```

This code has the same effect as choosing `Emphasis` in the Character Catalog. If no text is selected, the code has no effect.

Changing individual text properties

If you need to apply only an individual property to a text range, use `F_ApiSetTextProp()`. For example, the following code changes the font family of the selected text to AvantGarde:

```

. . .
#include "fstrings.h"
F_TextRangeT tr;
F_PropValT prop;
F_ObjHandleT docId;
UIntT i = 0;
F_StringsT strings;

docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tr = F_ApiGetTextRange(FV_SessionId, docId, FP_TextSelection);
if (!tr.beg.objId) return;

/* Get list of font families available in current session. */
strings = F_ApiGetStrings(0, FV_SessionId, FP_FontFamilyNames);

/* Find index of AvantGarde in list of families in session. */
for (i=0; i<strings.len &&
     !F_StrEqual("AvantGarde", strings.val[i]); i++);
if (i == strings.len) return; /* Font not found. */

/* Free the returned strings. */
F_ApiDeallocateStrings(&strings);

/* Set up property. Set it to the index for AvantGarde. */
prop.propIdent.num = FP_FontFamily;
prop.propVal.valType = FT_Integer;
prop.propVal.u.ival = i;

/* Apply the property to the text selection. */
F_ApiSetTextPropVal(docId, &tr, &prop);
. . .

```

Applying conditions to text

The API uses the text property `FP_InCond` to specify the conditions applied to a text location. `FP_InCond` specifies an array that includes the IDs of conditions that apply to the text location.

To apply conditions to a text range, set the `FP_InCond` property for the text range. For example, the following code applies the Comment condition to the selected text:

```

. . .
F_TextRangeT tr;
F_PropValT prop;
F_ObjHandleT docId, commentId;
F_IntsT condIds;

docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tr = F_ApiGetTextRange(FV_SessionId, docId, FP_TextSelection);
if (!tr.beg.objId) return;

/* Get the ID of the Comment condition. */
commentId = F_ApiGetNamedObject(docId, FO_CondFmt, "Comment");

/* Set up array of conditions (just one, in this case). */
condIds.val = (IntT*) &commentId;
condIds.len = 1;

/* Set up property. */
prop.propIdent.num = FP_InCond;
prop.propVal.valType = FT_Ints;
prop.propVal.u.isval = condIds;

/* Apply the property to the text selection. */
F_ApiSetTextPropVal(docId, &tr, &prop);
. . .

```

Applying Boolean conditional expressions

In a FrameMaker document, you show or hide text in a document based on a build expression. To do this, you need to create a Boolean conditional expression for the document. Every document contains an active build expression. Use the following APIs to get and set the active build expression in a document.

```
/* To create a active Conditional Boolean expression */
IntT state = True;
//Build Expression
StringT expressionValue = (StringT) "\"CondTag1\" or \"CondTag2
\"";
//Build Expression name
StringT expressionName = (StringT)"customExpression";
/* Set the FP_BooleanConditionState property to true.
Required to ensure that the active Build Expression is used
This is equivalent to the Show as per Expression radio button on
the Show / Hide Conditional Text pod */
F_ApiSetInt(FV_SessionId, docId, FP_BooleanConditionState,
state);

/* Set the name of the Build Expression */
F_ApiGetString(FV_SessionId, docId,
FP_BooleanConditionExpressionTag, expressionName);
/* Set the value of the Build Expression */
F_ApiSetString(FV_SessionId, docId,
FP_BooleanConditionExpression, expressionValue);

/* To update the current active Conditional Boolean expression
*/
/*Note: In this example, we are not setting the Build Expression
name because we are updating the existing active Expression */
IntT state = True;
StringT expressionValue = (StringT) "\"CondTag1\" or \"CondTag2
\"";

F_ApiSetInt(FV_SessionId, docId, FP_BooleanConditionState,
state);
F_ApiSetString(FV_SessionId, docId,
FP_BooleanConditionExpression, expressionValue);

/* To get the Conditional Boolean expression property
information*/

/* Get FP_BooleanConditionState property value. This is
equivalent to checking the state of the Show as per Expression
radio button on the Show / Hide Conditional Text pod */
```

```
IntT state = F_ApiGetInt(FV_SessionId, docId,  
FP_BooleanConditionState);  
/* Get the Build expression */  
StringT expressionValue = F_ApiGetString(FV_SessionId, docId,  
FP_BooleanConditionExpression);  
/* Get the name of the Build expression */  
StringT expressionName = F_ApiGetString(FV_SessionId, docId,  
FP_BooleanConditionExpressionTag);
```

Setting type-in properties

A Frame document has a set of properties called *type-in properties*, which specify the text characteristics of the insertion point in the document. Type-in properties do not apply to text that is already in a document; they apply only to text as the user types it in the document.

Whenever the user changes the insertion point in a document, the FrameMaker product sets the document's type-in properties to match the text properties of the character to the left of the insertion point. However, a document's type-in properties do not have to match the text properties of the character to the left of the insertion point. You can change them with the `F_ApiSetPropertyType()` and `F_ApiSetProps()` functions. For example, the following code sets the active document's type-in properties so that text the user types appears in uppercase and is underlined:

```
. . .  
F_ObjHandleT docId;  
  
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);  
F_ApiSetInt(FV_SessionId, docId,  
            FP_Capitalization, FV_CAPITAL_CASE_UPPER);  
F_ApiSetInt(FV_SessionId, docId,  
            FP_Underlining, FV_CB_SINGLE_UNDERLINE);  
. . .
```

Executing Clipboard functions

The API provides functions that programmatically execute Frame Clipboard functions. It also provides a Clipboard stack, which allows you to save the Clipboard contents.

Executing Frame Clipboard functions

The API provides the following functions to programmatically execute Frame Clipboard functions:

- `F_ApiCut()` cuts the current selection to the Clipboard.
- `F_ApiCopy()` copies the current selection to the Clipboard.
- `F_ApiPaste()` pastes Clipboard contents to the insertion point or current selection.
- `F_ApiClear()` clears the current selection.

All these functions work only on the active document. They use the Frame Clipboard and the current selection (or insertion point) in a document. They work with text, table cells, and graphic objects. You can allow the user to set the selection or insertion point, or you can do it programmatically.

The syntax for the functions is:

```
IntT F_ApiCut(F_ObjHandleT docId,
             IntT flags);
```

```
IntT F_ApiCopy(F_ObjHandleT docId,
              IntT flags);
```

```
IntT F_ApiPaste(F_ObjHandleT docId,
               IntT flags);
```

```
IntT F_ApiClear(F_ObjHandleT docId,
               IntT flags);
```

This argument	Means
docId	The ID of the document in which you want to cut, copy, paste, or clear the selection.
flags	Bit flags that specify how to cut, copy, paste, or clear the selection. See the table below. Specify 0 for the default behavior.

Specifying 0 for `flags` instructs these functions to behave in the following ways:

- `F_ApiCut()` and `F_ApiClear()` leave selected table cells empty and delete hidden text.
- `F_ApiPaste()` inserts table columns to the left of the current columns and rows above the current row.
- All functions suppress any Frame dialog boxes or alert boxes that arise.

To specify the behavior of the functions, you can OR the following values into the `flags` argument.

This value	Means	Applies to
FF_INTERACTIVE	Prompt user with dialog boxes or alert boxes that arise.	All Clipboard functions
FF_CUT_TBL_CELLS	Remove cut or cleared table cells.	<code>F_ApiClear()</code> and <code>F_ApiCut()</code>

This value	Means	Applies to
FF_DONT_DELETE_HIDDEN_TEXT	Don't delete hidden text.	F_ApiClear(), F_ApiCut(), and F_ApiPaste()
FF_INSERT_BELOW_RIGHT	Add columns to the right of the current column or below the current row.	F_ApiPaste()
FF_VISIBLE_ONLY	Cut, copy, paste, or clear only the visible portion of the selection.	All Clipboard functions
FF_REPLACE_CELLS	Replace selected cells with cells on the Clipboard.	F_ApiPaste()
FF_DONT_APPLY_ALL_ROWS	Don't apply condition setting on the Clipboard to all rows. If whole table is selected and Clipboard contains condition setting, cancel the paste.	F_ApiPaste()

The `FF_INTERACTIVE` flag takes precedence over other flags. So, if you specify `FF_INTERACTIVE | FF_DONT_DELETE_HIDDEN_TEXT` and the selection contains hidden text, the FrameMaker product prompts the user and allows the user to choose whether to delete the hidden text.

Saving the Clipboard contents

In some cases, you may want to use the Clipboard and then restore its original contents when you are done. The API provides a Clipboard stack, which allows you to do this. To manipulate the Clipboard stack, use `F_ApiPushClipboard()` and `F_ApiPopClipboard()`. The syntax for these functions is:

```
IntT F_ApiPushClipboard(VoidT);
IntT F_ApiPopClipboard(VoidT);
```

`F_ApiPushClipboard()` pushes the current Clipboard contents onto the Clipboard stack. `F_ApiPopClipboard()` pops the set of Clipboard contents on the top of the Clipboard stack to the Clipboard.

For example, the following code executes Copy and Paste operations and then restores the original Clipboard contents:

```
. . .  
F_ObjHandleT docId;  
  
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);  
F_ApiPushClipboard();  
F_ApiCopy(docId, 0);  
F_ApiPaste(docId, 0);  
F_ApiPopClipboard();  
. . .
```

Manipulating Asian Text

.....

.....

This chapter describes how to use the API to manipulate Asian text in Frame documents. Specifically, it discusses:

- Creating a rubi group
- The text encodings supported by FrameMaker products
- How to use encoding data in an API client
- Inspecting and manipulating encoded text
- Parsing encoded strings
- Getting the encoding for a text item
- Special issues such as decimal tabs, character offsets, and internal strings

To better understand the material in this chapter, you may want to learn more about how the API represents text. For information on this subject, see “Text” on page 114.

Creating a rubi group

Users create rubi groups by selecting the oyamoji text and choosing Rubi from the Special menu. Internally, a rubi group is stored as an anchored object. To create a rubi group via the API, use `F_ApiNewAnchoredObject ()`.

Note that `F_ApiNewAnchoredObject ()` accepts a text location, but not a selection as an argument. This changes the process for creating a group slightly:

- Identify the location for the rubi group
- Get the oyamoji location
- Insert the oyamoji text
- Get the rubi location
- Insert the rubi text

Once you have a rubi group, you can get the text range for the oyamoji and rubi text. In this way, you can edit the rubigroup contents. The following example shows how to create a new rubi group:

```

. . .
F_TextRangeT tr, oyamojiTr, rubiTr;
F_ObjHandle docId, rubiGrpId;
StringT s1, s2;

/* Assuming known contents in s1 and Hiragana chars in s2... */

docId = F_ApiGetID(0, FV_SessionId, FP_ActiveDoc);
tr = F_ApiGetTextRange(FV_SessionId, docId, FP_TextSelection);
if (tr.beg.objId == 0) return;

/* Insert the Rubi Group */
rubiGrpId = F_ApiNewAnchoredObject(docId, FO_Rubi, &tr.beg);
/* Get the location of the oyamoji and add the text. */
oyamojiTr = F_ApiGetTextRange(docId, rubiGrpId,
FP_OyamojiTextRange);
F_ApiAddText(docId, &oyamogiTr.beg, s1);
/* Get the location of the rubi and add the text. */
rubiTr = F_ApiGetTextRange(docId, rubiGrpId, FP_RubiTextRange);
F_ApiAddText(docId, &rubiTr.beg, s2);
. . .

```

Text encodings

Asian character sets include thousands of characters, and so the single byte range used for Roman character sets is insufficient. Asian character sets use single and double byte codes to identify each character. They also reserve the same single byte encodings for nearly every character in the 7-bit ASCII range. Any differences among characters in the ASCII range are relatively insignificant.

For example, Shift-JIS (an encoding for Japanese characters) replaces the ASCII characters "|", "~", and "\" with a solid vertical bar, and overbar, and the Yen symbol, respectively. Pathnames using the "\" character in a Roman encoding would appear with

the Yen symbol in Shift-JIS. However, the pathname would still parse correctly because the character codes are the same.

For text that must be shared across platforms and encodings, it is best to stay within the 7-bit ASCII range. Filenames are a good example of text that should follow this rule. Also, keeping within 7-bit ASCII for tag names (paragraph format tags, character format tags, etc.) is a good way to ensure a document will be usable on systems that support different languages.

Encoding schemes

There are a number of encoding schemes that map the codes to individual characters. Not only are there different encodings for each language, but within a language there might be a number of encodings. FrameMaker products can save and import text in a number of encodings:

Language	Encodings
Roman	FrameRoman ISOLatin-1 ASCII ANSI (Windows) Macintosh ASCII
Japanese	Shift-JIS JIS EUC
Traditional Chinese	Big5 EUC-CNS
Simplified Chinese	GB HZ
Korean	KSC8

Of these encodings, FrameMaker products use the following to represent characters internally. The following strings are the names Framemaker products use to identify these internal encodings:

Language	FrameMaker internal encoding names
Roman	FrameRoman
Japanese	JISX0208.ShiftJIS
Traditional Chinese	Big5
Simplified Chinese	GB2312-80.EUC
Korean	KSC5601-1992

FrameMaker Roman encoding

8-bit Roman character sets all share the same 7-bit ASCII characters. FrameMaker products also use the characters in the x80 - xFF range for special characters such as

non-breaking hyphens or em spaces. Asian fonts cannot be expected to support the same special characters. As a result, if the `FP_DialogEncodingName` is set to an Asian encoding, the user cannot type these special characters in text boxes or other parts of the user interface that are controlled by this setting. However, the user can type these characters in document text whenever the current font is a Roman font that supports them.

Using encoding data

The FDE provides functions to operate on strings and characters of a specified internal encoding. To use font encoding information, you must first initialize the font encoding data. Then you can get the internal encoding for a given character or string, set the encoding, or convert from the FrameMaker internal encoding to a different one; from Shift-JIS to EUC, for example.

Initializing encoding data and setting the U/I encoding

The API uses `F_FdeInitFontEncs()` to:

- initialize the font encoding data
- set the encoding for your client's user interface.

Initializing the encoding data sets up structures to represent each internal encoding supported by the current release of the FrameMaker product. This is true even if the current session doesn't support the languages those encodings represent. For example, you could process a string of Korean text with the FDE, even if the current session of FrameMaker would not be able to display it correctly.

The typical way to use this function is to initialize the FDE and then get the encoding name used for the current FrameMaker session. Then you pass that encoding name to `F_FdeInitFontEncs()` so your client will use the same encoding for its dialog boxes.

The syntax for `F_FdeInitFontEncs()` is:

```
FontEncIdT F_FdeInitFontEncs(ConStringT fontEncName);
```

This argument	Means
<code>fontEncName</code>	The name of the font encoding to use for your client's dialog boxes.

Possible values for `fontEncName` are:

Value	Means
FrameRoman	Roman text
JISX0208.ShiftJIS	Japanese text
BIG5	Traditional Chinese text
GB2312-80.EUC	Simplified Chinese text
KSC5601-1992	Korean text

The returned `FontEncIdT` is the ID of the font encoding you specified for your dialog boxes.

Example

The following code initializes the FDE and ensures the dialog box encoding is one the client can support. If the dialog box encoding for the current session is Japanese or Simplified Chinese, it passes that encoding to `F_FdeInitFontEncs()`. Otherwise, it passes Roman to `F_FdeInitFontEncs()`:

```

. . .
FontEncIdT feId;
StringT encName;

F_FdeInit();
encName = F_ApiGetString(0, FV_SessionId,
FP_DialogEncodingName);
if (F_StrIEqual(encName, "JISX0208.ShiftJIS") ||
    F_StrIEqual(encName, "GB2312-80.EUC"))
    feId = F_FdeInitFontEncs((ConStringT) encName);
else
    feId = F_FdeInitFontEncs((ConStringT) "FrameRoman");
. . .

```

Getting the encoding for fonts

Font families and individual fonts have associated encodings. The possible encodings are:

Value	Means
FrameRoman	Roman text
JISX0208.ShiftJIS	Japanese text
BIG5	Traditional Chinese text
GB2312-80.EUC	Simplified Chinese text
KSC5601-1992	Korean text
Multiple	More than one encoding for the font family

If the returned encoding is `Multiple`, the font family has different encodings for its different variations. In that case, you must get the encoding for each variation. Non-text fonts may return `FrameRoman`, or they may return the family name of the font. For example, on some platforms the encoding for the Symbol font family is indicated by the string `Symbol`.

Getting the encoding for a font family

To get the encoding for a font family, first get a list of font families, then loop through that list to get the index of the family you want. Then you pass the index to `F_ApiGetEncodingForFamily()`.

The syntax for `F_ApiGetEncodingForFamily()` is:

```
StringT F_ApiGetEncodingForFamily(IntT family);
```

This argument	Means
family	The font family for which you want to know the encoding.

Example

The following code gets the index for the Minchu font family from the session list of font families. It then gets the encoding for that font family:

```

. . .
#include "futils.h"
#include "fstrings.h"
#include "fencode.h"
. . .
F_StringsT families;
StringT encoding;
UIntT i;
/* First get the list of font families for the session */
families = F_ApiGetStrings(0, FV_SessionId, FP_FontFamilyNames);
/* Now get the index of the Minchu family */
for (i=0; i < families.len; i++)
    if (F_StrIEqual(families.val, (StringT) "minchu")) break;
if (i == families.len) return; /* Minchu not found */
/* Now use the index to get the encoding for Minchu */
encoding = F_ApiGetEncodingForFamily(i);
. . .
/* Free the strings */
F_ApiDeallocateStrings(&families);
F_ApiDeallocateString(&encoding);

```

Getting the encoding for a font variation

If the font family returns an encoding of `Multiple`, you should use `F_ApiFamilyFonts()` to get a list of the variations for the family. Then you can use `F_ApiGetEncodingForFont()` to get the encoding for a specific variation.

The syntax for `F_ApiGetEncodingForFont()` is:

```
StringT F_ApiGetEncodingForFont(F_FontT *font);
```

This argument	Means
font	Pointer to a structure listing the font's name, weight, angle, and variation

Example

The following code loops through the session fonts, then loops through the permutations of each and prints the encoding for each permutation to the console:

```

. . .
F_FontsT fam;
F_StringsT families, weights, variations, angles;
StringT encoding;
UIntT i, j;

/* Get lists of families, variations, weights, and angles. */
families = F_ApiGetStrings(0, FV_SessionId, FP_FontFamilyNames);
weights = F_ApiGetStrings(0, FV_SessionId, FP_FontWeightNames);
variations = F_ApiGetStrings(0, FV_SessionId,
                             FP_FontVariationNames);
angles = F_ApiGetStrings(0, FV_SessionId, FP_FontAngleNames);

/* Loop through each session font */
for (i=0; i < families.len; i++) {
/* Now print the encoding for each variation to the console */
    fam = F_ApiFamilyFonts(i);
    for (j = 0; j < fam.len; j++) {
        encoding = F_ApiGetEncodingForFont(fam.val[j]);
        F_Printf(NULL, "The encoding for %s-%s-%s-%s is %s\n"
                families.val[fam.val[j].family],
                weights.val[fam.val[j].weight],
                variations.val[fam.val[j].variation],
                angles.val[fam.val[j].angle],
                encoding);
        F_ApiDeallocateString(&encoding);
    }
}
/* Be sure to free the structures and strings */
. . .

```

Determining which encodings are currently supported

The API has two functions to determine which encodings are supported for the current session. `F_ApiIsEncodingSupported()` returns `True` if the passed encoding is currently supported. `F_ApiGetSupportedEncodings()` returns a `F_StringsT` list of all the encodings supported for the current session.

The syntax for `F_ApiIsEncodingSupported()` is:

```
BoolT F_ApiGetEncodingForFamily(ConStringT encodingName);
```

This argument	Means
<code>encodingName</code>	The encoding of interest. Possible values are: <code>FrameRoman</code> <code>JISX0208.ShiftJIS</code> <code>BIG5</code> <code>GB2312-80.EUC</code> <code>KSC5601-1992</code>

The syntax for `F_ApiGetSupportedEncodings()` is:

```
F_StringsT F_ApiGetSupportedEncodings();
```

Inspecting and manipulating encoded text

For text of a given encoding, you can perform actions such as string comparison, search for the occurrence of a character in a string, character count, truncation, concatenation, and others. The functions to perform these actions are much like the corresponding string functions for Roman text. However, they must be passed an encoding ID so they can check a character code against the encoding.

For example, a single byte code might be a single byte character in one encoding, while in another encoding it might be the first or last byte of a double byte character. The API provides functions to determine exactly that.

Getting encoding IDs

When you initialize the encoding data, the FDE assigns an ID to each encoding data structure. String functions that use this encoding data generally require the ID to identify the encoding.

The FDE includes functions to get the encoding ID assigned to an encoding name, and to get the encoding name that is associated with a given encoding ID.

`F_FontEncId()` returns the `FontEncIdT` for the encoding data associated with the specified encoding name. If the encoding name is not supported for the current session, this function returns the ID for the `FrameRoman` encoding.

The syntax for `F_FontEncId()` is:

```
FontEncIdT F_FontEncId(ConStringT fontEncName);
```

This argument	Means
<code>fontEncName</code>	The encoding of interest. Possible values are: <code>FrameRoman</code> <code>JISX0208.ShiftJIS</code> <code>BIG5</code> <code>GB2312-80.EUC.</code> <code>KSC5601-1992</code> <code>Multiple</code>

`F_FontEncName()` returns the encoding name associated with the specified `FontEncIdT`. If the specified `FontEncIdT` is not valid, this function returns a `NULL` string.

The syntax for `F_FontEncName()` is:

```
ConStringT F_FontEncName(FontEncIdT fontEncId);
```

This argument	Means
<code>fontEncId</code>	The encoding ID of interest

Functions for encoded strings

The FDE includes the following functions for handling strings of a given encoding. For more information, see these functions in the *FDK Programmers Reference* guide.

```
F_StrChrEnc()  
F_StrRChrEnc()  
F_StrStrEnc()  
F_StrIEqualEnc()  
F_StrIEqualNEnc()  
F_StrICmpEnc()  
F_StrCmpNEnc()  
F_StrICmpNEnc()  
F_StrTruncEnc()  
F_StrLenEnc()  
F_StrCatDblCharNEnc()  
F_StrIPrefixEnc()  
F_StrISuffixEnc()  
F_StrCatNEnc()  
F_StrNCatNEnc()  
F_StrCpyNEnc()
```

Parsing an encoded string

For the characters in a given string, you might need to know whether a character is single byte, whether a single byte is the first or last byte of a double byte character, or whether two consecutive bytes comprise a valid double byte character. The API has the following functions that map the given byte or bytes to the specified encoding:

```
BoolT F_CharIsDoubleByteFirst(UCharT char, FoneEncIdT feId);  
BoolT F_CharIsDoubleByteSecond(UCharT char, FoneEncIdT feId);  
BoolT F_CharIsDoubleByte(UCharT firstChar,  
                          UCharT secondChar, FoneEncIdT feId);
```

Example

The following code checks each character in a string to see whether it is one byte or two and increments by the correct amount:

```
. . .
StringT dbEncString;
UCharT currChar;
IntT i = 0;
FontEncIdT feId;
feId = F_FontEncId((ConStringT) "JISX0208.ShiftJIS");
. . .
/* Assume there is a Japanese string in dbEncString...
 * We do not need to also check if the second byte is null,
 * as all of the supported encodings do not have '\0' in their
 * valid range.
 */
while (dbEncString[i] != '\0') {
    if (F_CharIsDoubleByte(dbEncString[i], dbEncString[i+1],
feId))
    {
        /* This is a double byte character... */
        i = i + 2;
    }
    else {
        /* This is either a true single byte char,
 * or the second byte was not a double-byte second,
 * so treat as a single char for proper scanning.
 */
        i++;
    }
}
}
```

Getting the encoding for a text item

The function `F_ApiGetText()` returns a `F_TextItemsT` structure, which is a list of text items that makes up a range of document text. Note that among other things, a single text item can represent a string of characters with common text properties. For a change in character encoding to occur, there must be a corresponding change in some text property such as a new font family or font variation. This means that any text item that is a string must be a string of a given encoding. For more information about text items, see “Text” on page 114.

For a given text item, you can use the offset to determine a text location corresponding to that text item. You can then use `F_ApiGetTextPropVal()` to get the font at that location. Given the font, you can use `F_ApiGetEncodingForFont()` to determine the encoding at that location. You now know the encoding for a given text item.

As you scan the items in a `F_TextItemsT` structure, you can flag changes to the text encoding. Until you see a change in text encoding, you can assume any string text items are of the current encoding.

Keep in mind that a change of text encoding necessarily occurs at a change in character properties. This is indicated in the `F_TextItemT` as a data type of `FTI_CharPropsChange`. The data for an `FTI_CharPropsChange` is a flag to indicate the type of change. If the flag indicates `FTF_ENCODING`, you know the encoding has changed, and you must get the encoding for the next string text item. For more information, see “`FTI_CharPropsChange`” on page 117 and “`FTF_ENCODING`” on page 119.

Special issues with double byte encodings

Following are some special issues to keep in mind when working with double byte text.

Decimal tabs

The `F_TabT` data structure describes an individual tab stop. It includes a field for a character (such as a period or a comma) for decimal-aligned tab stops. `FrameMaker` products only support single byte characters to align tab stops.

Offsets into strings

Unless otherwise noted in the *FDK Programmers Reference*, string functions that return an offset into the string express the offset in terms of characters, and not bytes. This should keep your existing code viable. For example, code to set text locations and text ranges should still work, even for text that contains double byte text.

To get the count of characters in an encoded string, use `F_StrLenEnc()`. This function returns the number of characters in the string, even though some characters might be single byte and others might be double byte. (Remember that double byte encodings reserve single byte space for certain characters.) The syntax for `F_StrLenEnc()` is:

```
IntT F_StrLenEnc(ConStringT s, FontEncIdT feId);
```

This argument	Means
<code>s</code>	The string whose characters you want to count
<code>feId</code>	The ID of the encoding for <code>s</code>

Internal strings in FrameMaker products

Internal strings such as encoding names, marker text, or FrameMaker product client names all use text in the 7-bit ASCII range. Tags (paragraph format tags, for example) can use double byte text. However, for FrameMaker documents the tag names of elements cannot use double byte text.

Text in an unsupported encoding

A document can include text in an encoding that is not supported by the current system configuration. In this case, FrameMaker retains the encoding identification with the text even though it can't display the text correctly. This is referred to as a ghost encoding.

For example, suppose a document contains Japanese text, but the system can only display Western text. The Japanese text appears as a series of arbitrary characters, each character corresponding to a single byte of what might be a double-byte or single-byte Japanese character. If you get the encoding for that text, the FDE will return `JISX0208.ShiftJIS`.

Note that ghost encoded text is displayed as though it is FrameRoman text. This is important because it is likely that line breaks will split up double-byte characters. Furthermore, functions like `F_CharIsDoubleByteFirst()` and `F_CharIsDoubleByteSecond()` return unreliable results. For this reason, you have no way to ensure ghost encoded text items are valid, and we suggest you specifically do not process any text that uses ghost encodings.

Creating and Deleting API Objects

.....

⋮

To create or destroy anything in a FrameMaker product document, you must create or destroy the object the API uses to represent it. This chapter discusses how to create and destroy objects.

Before you use API functions to create and delete objects, you need an understanding of how the Frame API organizes objects. For background information on this subject, see Part II, “Frame Product Architecture.”

Creating objects

The API provides different functions for creating different types of objects. For example, you use `F_ApiNewTable()` to create tables and `F_ApiNewSeriesObject()` to create objects that occur in ordered series. The following table lists the API object types and the functions you use to create them.

To create objects of this type	Use
<code>FO_Book</code>	<code>F_ApiNewNamedObject()</code>
<code>FO_CharFmt</code>	
<code>FO_Color</code>	
<code>FO_CondFmt</code>	
<code>FO_ElementDef</code>	
<code>FO_FmtChangeList (named)</code>	
<code>FO_MasterPage</code>	
<code>FO_MenuItemSeparator</code>	
<code>FO_PgfFmt</code>	
<code>FO_RefPage</code>	
<code>FO_RulingFmt</code>	
<code>FO_TblFmt</code>	
<code>FO_VarFmt</code>	
<code>FO_XRefFmt</code>	

To create objects of this type	Use
FO_Arc	F_ApiNewGraphicObject()
FO_Ellipse	
FO_Flow ^a	
FO_Group	
FO_Inset	
FO_Line	
FO_Math	
FO_Polyline	
FO_Polygon	
FO_Rectangle	
FO_RoundRect	
FO_TextFrame	
FO_TextLine	
FO_UnanchoredFrame	
FO_AFrame	F_ApiNewAnchoredObject()
FO_Fn	
FO_Marker	
FO_Rubi	
FO_Tbl	
FO_TiApiClient	
FO_BodyPage	F_ApiNewSeriesObject()
FO_BookComponent	
FO_Pgf	
FO_BookComponent	F_ApiNewBookComponentInHierarchy()
FO_Element	F_ApiNewElement() ^b F_ApiNewElementInHierarchy()
FO_FmtRule	F_ApiNewSubObject()
FO_FmtRuleClause	
FO_FmtChangeList (unnamed)	
FO_Tbl	F_ApiNewTable()
FO_Tbl	F_ApiNewAnchoredFormattedObject()
FO_Var	
FO_XRef	

To create objects of this type	Use
FO_Cell	F_ApiAddCols() F_ApiAddRows()
FO_Row	F_ApiAddRows()
FO_Inset FO_TiFlow FO_TiText FO_TiTextTable	F_ApiImport()
FO_Command	F_ApiDefineCommand() F_ApiDefineAndAddCommand()
FO_Menu	F_ApiDefineMenu() F_ApiDefineAndAddMenu()
FO_Doc ^c	F_ApiOpen() F_ApiSimpleNewDoc() F_ApiCustomDoc()

- To create a flow, you must create a text frame. See “Creating flows” on page 370.
- You can also create new elements with `F_ApiWrap()` and `F_ApiSplitElement()`.
- For information on creating documents and books, see Chapter 4, “Executing Commands with API Functions.”

If they succeed, these functions return the ID of the object they create. If they fail, they return 0 and assign an error code to `FA_errno`.

Creating named objects

A *named object* is an object, such as a master page or a Paragraph Catalog format, that is identified by a unique name. To create named objects, use

`F_ApiNewNamedObject()`.

The syntax for `F_ApiNewNamedObject()` is:

```
F_ObjHandleT F_ApiNewNamedObject(F_ObjHandleT docId,
    IntT objType,
    StringT name);
```

This argument	Means
docId	The ID of the document to which to add the object. To create a book, specify <code>FV_SessionId</code> .

This argument	Means
objType	The type of object to create (for example, FO_MasterPage, FO_PgfFmt, or FO_Book).
name	The name to give to the object. If the object is an FO_Book object, specify the pathname of the book file to create.

F_ApiNewNamedObject () uses a set of default properties when it creates a new named object. Because the property lists for most named objects are quite long, it is often easier to copy the properties from a similar object and then change individual properties.

Creating a paragraph format

The following code creates a paragraph format named MyHead, which looks like Heading1 except that it's indented two inches:

```
. . .
#define in (MetricT)(65536*72)
F_PropValsT proplist;
F_ObjHandleT docId, Heading1Id, myHeadId;

docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);

/* Create MyHead paragraph format. */
myHeadId = F_ApiNewNamedObject(docId, FO_PgfFmt, "MyHead");

/* Get ID for Heading1. */
Heading1Id = F_ApiGetNamedObject(docId, FO_PgfFmt, "Heading1");
if (!Heading1Id) return;

/* Get properties for Heading1 and copy them to MyHead. */
proplist = F_ApiGetProps(docId, Heading1Id);
F_ApiSetProps(docId, myHeadId, &proplist);

F_ApiSetMetric(docId, myHeadId, FP_LeftIndent, 2*in);
F_ApiDeallocatePropVals(&proplist);

. . .
```

Creating a book

The following code creates a book named `mybook` in the `tmp` directory. It uses `F_ApiNewSeriesObject()` to add book components. For more information on `F_ApiNewSeriesObject()`, see “Creating series objects” on page 369. For more information on creating books and book components, see “Creating new books and components” on page 166.

```

. . .
F_ObjHandleT bkId, componentId;

bkId = F_ApiNewNamedObject(FV_SessionId, FO_Book,
                          "/tmp/mybook");

/* Create component and then change its name. */
componentId = F_ApiNewSeriesObject(bkId,
                                   FO_BookComponent, 0);
F_ApiSetString(bkId, componentId, FP_Name, "doc1");

/* Add another component after the first one. */
componentId = F_ApiNewSeriesObject(bkId,
                                   FO_BookComponent, componentId);
F_ApiSetString(bkId, componentId, FP_Name, "doc2");
. . .

```

Creating graphic objects

To create any graphic object except an anchored frame, use `F_ApiNewGraphicObject()`.

The syntax for `F_ApiNewGraphicObject()` is:

```

F_ObjHandleT F_ApiNewGraphicObject(F_ObjHandleT docId,
                                   IntT objType,
                                   F_ObjHandleT parentFrameId);

```

This argument	Means
<code>docId</code>	The ID of the document in which to create the new object
<code>objType</code>	The type of graphic object to create (for example, <code>FO_Rectangle</code> or <code>FO_Line</code>)
<code>parentFrameId</code>	The ID of the parent frame in which to create the object

You can create a graphic object only in a frame. To create a graphic object directly on a page (not in an anchored or unanchored frame), you create it in the *page frame*. A page frame is an invisible frame that every page has. For more information on page frames and how FrameMaker products organize graphics, see “How the API represents graphic objects” on page 93.

The API maintains a frame’s *child objects* in a linked list. The order of this list corresponds to the back-to-front draw order. If the frame you specify for `parentFrameId` already has child objects, `F_ApiNewGraphicObject()` adds the new object to the end of the linked list; that is, it puts it in front of the other objects in the frame. The API automatically updates the properties of the parent frame and the last object in the list to reflect the addition of the new object. For instructions on moving objects forward or back in the draw order, see “Moving graphics forward or back in the draw order” on page 304.

`F_ApiNewGraphicObject()` uses a set of arbitrary default values for the properties of the graphic objects that it creates. Usually, you will need to change most of these default properties.

Example

To draw a circle with a one-inch diameter directly on the current page of a document, use the following code:

```
. . .
#define in (MetricT) (65536*72)
F_ObjHandleT docId, pageId, pFrameId, circleId;

/* Get the document, current page, and page frame IDs.
** The F_ApiGet[property_type]() and F_ApiSet[property_type]()
** functions are explained in Chapter 5.
*/
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
pageId = F_ApiGetId(FV_SessionId, docId, FP_CurrentPage);
pFrameId = F_ApiGetId(docId, pageId, FP_PageFrame);

/* Create the circle on the page frame. */
circleId = F_ApiNewGraphicObject(docId, FO_Ellipse, pFrameId);

/* Change the size of the circle to 1 inch. */
F_ApiSetMetric(docId, circleId, FP_Height, in);
F_ApiSetMetric(docId, circleId, FP_Width, in);

/* Move the circle toward the center of the page. */
F_ApiSetMetric(docId, circleId, FP_LocX, 2*in);
F_ApiSetMetric(docId, circleId, FP_LocY, 3*in);
. . .
```

Creating anchored objects

An *anchored object* is an object, such as a cross-reference, table, or anchored frame, that the user can insert in text. The API provides the following functions to create anchored objects:

- To create tables, use `F_ApiNewTable()`.
- To create variables and cross-references, use `F_ApiNewAnchoredFormattedObject()`.
- To create other anchored objects, use `F_ApiNewAnchoredObject()`.

`F_ApiNewTable()` is discussed in “Creating tables” on page 376.

The syntax for `F_ApiNewAnchoredFormattedObject()` and `F_ApiNewAnchoredObject()` is:

```
F_ObjHandleT F_ApiNewAnchoredFormattedObject(F_ObjHandleT docId,
      IntT objType,
      StringT format,
      F_TextLocT *textLocp);
```

```
F_ObjHandleT F_ApiNewAnchoredObject(F_ObjHandleT docId,
      IntT objType,
      F_TextLocT *textLocp);
```

This argument	Means
<code>docId</code>	The ID of the document to which to add the object
<code>objType</code>	The type of object to create (for example, <code>FO_Marker</code> or <code>FO_XRef</code>)
<code>format</code>	The string that specifies the object’s format (for example, <code>Heading & Page</code> or <code>Current Date (Long)</code>)
<code>textLocp</code>	The text location at which to insert the anchored object

`F_ApiNewAnchoredObject()` and `F_ApiNewAnchoredFormattedObject()` use a set of arbitrary default properties to create new objects. For example, the default width and height of a new anchored frame is 0.25 inches.

Example

The following code adds a `Filename (Long)` variable at the insertion point (or the beginning of the text selection) of the active document:

```

. . .
F_TextRangeT tr;
F_ObjHandleT docId, variableId;

/* Get the insertion point. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tr = F_ApiGetTextRange(FV_SessionId, docId, FP_TextSelection);

/* Return if there is no selection or IP. */
if(!tr.beg.objId) return;

/* Insert the variable. */
variableId = F_ApiNewAnchoredFormattedObject(docId, FO_Var,
                                             "Filename (Long)", &tr.beg);
. . .

```

Creating series objects

A *series object* is any object, other than a graphic object, that occurs in an ordered series. Paragraphs and body pages are examples of series objects. To create a series object, use `F_ApiNewSeriesObject()`.

The syntax for `F_ApiNewSeriesObject()` is:

```

F_ObjHandleT F_ApiNewSeriesObject(F_ObjHandleT docId,
    IntT objType,
    F_ObjHandleT prevId);

```

This argument	Means
<code>docId</code>	The ID of the document or book to which to add the object.
<code>objType</code>	The type of object to create (for example, <code>FO_BodyPage</code> or <code>FO_Pgf</code>).
<code>prevId</code>	The ID of the object that you want to add the new object after. To add a paragraph at the beginning of a flow, specify the flow's ID. To add other objects at the beginning of a series, specify 0.

Example

The following code inserts a paragraph after the paragraph containing the insertion point:

```

. . .
F_ObjHandleT docId, pgfId;
F_TextRangeT tr;
F_TextLocT textLoc;

/* Get the insertion point. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tr = F_ApiGetTextRange(FV_SessionId, docId, FP_TextSelection);
if (!tr.beg.objId) return;

/* Add the paragraph. */
pgfId = F_ApiNewSeriesObject(docId, FO_Pgf, tr.beg.objId);

/* Put some text in the paragraph. */
textLoc.objId = pgfId;
textLoc.offset = 0;
F_ApiAddText(docId, &textLoc, "Here's some text");
. . .

```

Creating flows

You can't create a flow directly with API functions. However, you can create one indirectly by creating a text frame with `F_ApiNewGraphicObject()`. Each time you create a text frame, the API automatically creates a flow to contain it. For example, to create a flow on the first body page of the active document, you can use the following code:

```

. . .
F_ObjHandleT docId, tFrameId, flowId, pageId, pFrameId;

/* Get ID of first body page's page frame. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
pageId = F_ApiGetId(FV_SessionId, docId, FP_FirstBodyPageInDoc);
tFrameId = F_ApiGetId(docId, pageId, FP_PageFrame);

/* Create the text frame and get the flow's ID from it. */
tFrameId = F_ApiNewGraphicObject(docId, FO_TextFrame, pFrameId);
flowId = F_ApiGetId(docId, tFrameId, FP_Flow);
. . .

```

To connect a text frame in a flow, set its `FP_NextTextFrameInFlow` or `FP_PrevTextFrameInFlow` property to the ID of a text frame that is already in the flow. To disconnect a text frame from a flow, set its `FP_NextTextFrameInFlow` or `FP_PrevTextFrameInFlow` property to 0.

Creating structural elements

The API provides the following functions for creating structural elements in FrameMaker documents and books:

- `F_ApiNewElement()` inserts a new element at a specified text location in a document.
- `F_ApiNewElementInHierarchy()` inserts a new element at a specified position in the element hierarchy of a document or book.

The syntax for `F_ApiNewElement()` is:

```
F_ObjHandleT F_ApiNewElement(F_ObjHandleT docId,
    F_ObjHandleT elemDefId,
    F_TextLocT *textLocp);
```

This argument	Means
<code>docId</code>	The ID of the document to which to add the element
<code>elemDefId</code>	The ID of the element definition for the new element
<code>textLocp</code>	The text location at which to insert the new element

The syntax for `F_ApiNewElementInHierarchy()` is:

```
F_ObjHandleT F_ApiNewElementInHierarchy(F_ObjHandleT docId,
    F_ObjHandleT elemDefId,
    F_ElementLocT *elemLocp);
```

This argument	Means
<code>docId</code>	The ID of the document or book to which to add the element
<code>elemDefId</code>	The ID of the element definition for the new element
<code>elemLocp</code>	The location at which the element is inserted

You can't use `F_ApiNewElementInHierarchy()` to add elements to an unstructured document. You must structure the document first by adding a root element with `F_ApiNewElement()`.

Examples

The following code adds a Para element at the insertion point:

```

. . .
F_ElementRangeT elemSelect;
F_ObjHandleT docId, elemId, paraEdefId;

/* Get ID of active document and the Para element definition. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
paraEdefId = F_ApiGetNamedObject(docId, FO_ElementDef, "Para");

/* Get current element selection in active document. */
elemSelect = F_ApiGetElementRange(FV_SessionId, docId,
                                   FP_ElementSelection);

if (elemSelect.beg.parentId == 0 || paraEdefId == 0) return;

/* Insert the new element. */
elemId = F_ApiNewElementInHierarchy(docId, paraEdefId,
                                   &elemSelect.beg);
. . .

```

The following code adds a highest-level element, named Appendix, to the main flow of the active document:

```

. . .
F_ObjHandleT docId, elemId, chapEdefId;
F_TextLocT tl;

/* Get IDs of document, main flow, and element definition. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
chapEdefId = F_ApiGetNamedObject(docId, FO_ElementDef,
                                  "Appendix");

/* Set up text location for beginning of main flow. */
tl.objId = F_ApiGetId(FV_SessionId, docId, FP_MainFlowInDoc);
tl.offset = 0;

/* Insert the new element. */
elemId = F_ApiNewElement(docId, chapEdefId, &tl);
. . .

```

Creating format rules, format rule clauses, and format change lists

To create format rules, format rule clauses, and unnamed format change lists in FrameMaker documents and books, use `F_ApiNewSubObject()`.

The syntax for `F_ApiNewSubObject()` is:

```
F_ObjHandleT F_ApiNewSubObject(F_ObjHandleT docOrBookId,  
    F_ObjHandleT parentId,  
    IntT property);
```

This argument	Means
<code>docOrBookId</code>	The ID of the document in which to create the new object
<code>parentId</code>	The ID of the object's parent object
<code>property</code>	The property of the parent object to associate with the new object

`F_ApiNewSubObject()` allows you to associate the new object with a specified property of its parent object. For example, you can create an `FO_FmtRule` object as the suffix format rule of an element definition or as a subformat rule of a format rule clause. For a complete list of the properties with which you can associate new format rule objects, see “`F_ApiNewSubObject()`” in the FDK Programmer's Reference guide.

The following code creates a prefix rule and adds it to the Quotation element definition so that the element definition appears as shown in Figure 8-1:

```

. . .
F_ObjHandleT docId, quoteEdefId, prefixRuleId, clauseId;

/* Get ID of Para element definition. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
quoteEdefId = F_ApiGetNamedObject(docId, FO_ElementDef,
                                "Quotation");

/* Add the prefix format rule. */
prefixRuleId = F_ApiNewSubObject(docId, quoteEdefId,
                                FP_PrefixRules);

/* Set the rule type. */
F_ApiSetInt(docId, prefixRuleId, FP_FmtRuleType,
            FV_CONTEXT_RULE);

/* Add rule clause to suffix rule. 322 is left quote. */
clauseId = F_ApiNewSubObject(docId, prefixRuleId,
                             FP_FmtRuleClauses);
F_ApiSetInt(docId, clauseId, FP_IsTextRange, True);
F_ApiSetString(docId, clauseId, FP_ElemPrefixSuffix, "\322");
. . .

```

Element (Container): Quotation
General rule: <Text>.
Text format rules

1. **In all contexts.**
 Text range.
 No additional formatting.

Prefix rules

1. **In all contexts**
 Prefix: “
 Text range.

Figure 8-1 *Quotation element definition*

To create a named format change list, use `F_ApiNewNamedObject()`. To add the format change list to a format rule clause, set the format rule clause object's `FP_FmtChangeListTag` property to the name of the change list. For example, the

following code creates the Code format change list shown in Figure 8-2 and adds it to the first format rule clause of the Para element definition's first text format rule:

```

. . .
F_ObjHandleT docId, changeListId;
F_ObjHandleT edefId;
UIntT i;
F_StringsT fonts;
F_IntsT rules, clauses;
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
changeListId = F_ApiNewNamedObject(docId, FO_FmtChangeList,
    "Code");

/*
** Get the index of the Courier font family in the list of
** fonts available in the current session.
*/
fonts = F_ApiGetStrings(0, FV_SessionId, FP_FontFamilyNames);
for (i=0; i<fonts.len &&
    !F_StrEqual("Courier", fonts.val[i]); i++);
if (i == fonts.len) return; /* Courier font not found. */

/* Add the FP_FontFamily property; set it to Courier. */
F_ApiSetInt(docId, changeListId, FP_FontFamily, i);

/* Add the FP_PairKern property. */
F_ApiSetInt(docId, changeListId, FP_PairKern, False);

/* Get ID of the first clause of the first text format rule. */
edefId = F_ApiGetNamedObject(docId, FO_ElementDef,
    "Para");
rules = F_ApiGetInts(docId, edefId, FP_TextFmtRules);
clauses = F_ApiGetInts(docId, rules.val[0], FP_FmtRuleClauses);

/* Add the Code format change list to the format rule clause. */
F_ApiSetString(docId, clauses.val[0],
    FP_FmtChangeListTag, "Code");
. . .

```



Figure 8-2 Code format change list

Creating tables

The API provides the following functions for creating tables:

- `F_ApiNewTable()` is usually easier to use because it allows you to specify a format and the number of rows and columns.
- `F_ApiNewAnchoredObject()` creates a table with a single column and a single body row.

For instructions on using `F_ApiNewAnchoredObject()`, see “Creating anchored objects” on page 368.

The syntax for `F_ApiNewTable()` is:

```
F_ObjHandleT F_ApiNewTable(F_ObjHandleT docId,
    StringT format,
    IntT numCols,
    IntT numBodyRows,
    IntT numHeaderRows,
    IntT numFooterRows,
    TextLocT *textLocp);
```

This argument	Means
<code>docId</code>	The ID of the document.
<code>format</code>	The table format tag (for example, <code>FormatA</code> or <code>Wide Table</code>). To use the default format, specify <code>NULL</code> . The default format is the format of the last table the user inserted.
<code>numCols</code>	The number of columns in the table. To use the default number of columns from the Table Catalog format, specify <code>-1</code> .
<code>numBodyRows</code>	The number of rows in the table. To use the default number of body rows from the Table Catalog format, specify <code>-1</code> .
<code>numHeaderRows</code>	The number of header rows in the table. To use the default number of header rows from the Table Catalog format, specify <code>-1</code> .
<code>numFooterRows</code>	The number of footer rows in the table. To use the default number of footer rows from the Table Catalog format, specify <code>-1</code> .
<code>textLocp</code>	The location at which to insert the new table. The location can't be within a footnote or a table.

If successful, `F_ApiNewTable()` returns the ID of the new `FO_Tbl` object. Otherwise, it returns `0` and sets `FA_errno` to an error code.

Example

The following code inserts the table shown in Figure 8-3:

```

. . .
F_ObjHandleT docId, pgfId, tblId, titlePgfId;
F_TextRangeT tr;
F_TextLocT textLoc;

/* Get the insertion point. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tr = F_ApiGetTextRange(FV_SessionId, docId, FP_TextSelection);
if (!tr.beg.objId) return;

/* Insert the table at the insertion point. */
tblId = F_ApiNewTable(docId, NULL, 3, 3, 0, 0, &tr.beg);

/* Return if IP in FO_Tbl or FO_Fn, and table wasn't created.*/
if (!tblId) return;

/* Get ID of paragraph in table's title. */
titlePgfId = F_ApiGetId(docId, tblId, FP_FirstPgf);

/* Put some text in the table's title. */
textLoc.objId = titlePgfId;
textLoc.offset = 0;
F_ApiAddText(docId, &textLoc, "My Table");
. . .

```

My Table		

Figure 8-3 Table created with *F_ApiNewTable()*

For an example of how to add text to table cells, see “Adding text to table cells” on page 335.

Adding table rows and columns

To add table rows or columns to an existing table, use these functions:

- *F_ApiAddCols()* to add table columns
- *F_ApiAddRows()* to add table rows

You can't add rows by changing the `FO_Tbl` object's `FP_TblNumCols` and `FP_TblNumRows` properties. These properties are read-only.

The syntax for `F_ApiAddCols()` is:

```
IntT F_ApiAddCols(F_ObjHandleT docId,
                 F_ObjHandleT tableId,
                 IntT refColNum,
                 IntT direction,
                 IntT numNewCols);
```

This argument	Means
<code>docId</code>	The ID of the document containing the table.
<code>tableId</code>	The ID of the table to which to add columns.
<code>refColNum</code>	The column at which to start adding columns. The columns are numbered from left to right starting with column 0.
<code>direction</code>	The direction in which to add columns. To add columns to the left of the reference column specify <code>FV_Left</code> . To add columns to the right, specify <code>FV_Right</code> .
<code>numNewCols</code>	The number of columns to add.

If successful, `F_ApiAddCols()` returns `FE_Success`. Otherwise, it returns an error code.

The syntax for `F_ApiAddRows()` is:

```
IntT F_ApiAddRows(F_ObjHandleT docId,
    F_ObjHandleT refRowId,
    IntT direction,
    IntT numNewRows);
```

This argument	Means
<code>docId</code>	The ID of the document containing the table.
<code>refRowId</code>	The ID of the row at which to start adding rows. The added rows will be the same type as this row. For example, if <code>refRowId</code> specifies a heading row, the added rows will also be heading rows.
<code>direction</code>	The direction in which to add rows. To add rows above the reference row, specify <code>FV_Above</code> . To add them below, specify <code>FV_Below</code> . For a list of the other constants you can specify for this parameter, see “ <code>F_ApiAddRows()</code> ” in the FDK Programmer’s Reference guide.
<code>numNewRows</code>	The number of rows to add.

If successful, `F_ApiAddRows()` returns `FE_Success`.

`F_ApiAddCols()` requires you to specify the number of the reference column, whereas `F_ApiAddRows()` requires you to specify the ID of the reference row. This is because, in Frame document architecture, rows are objects. Columns are just a way of referring to a set of cells. When you create a row, the API actually creates an `FO_Row` object to represent the row and an `FO_Cell` object to represent each cell in the row. When you create a column, the API just creates `FO_Cell` objects and adds them to existing `FO_Row` objects. For more information on how tables and cells are organized, see “How the API represents tables” on page 141.

Example

The following code adds a column to the right of the first column and two rows below the second row in a table:

```
. . .
F_ObjHandleT docId, tblId, row1Id, row2Id;

/* Get the document and table IDs. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tblId = F_ApiGetId(FV_SessionId, docId, FP_SelectedTbl);

/* Add the column. */
F_ApiAddCols(docId, tblId, 0, FV_Right, 1);

/* Get the ID for row 1, then traverse to the ID for row 2. */
row1Id = F_ApiGetId(docId, tblId, FP_FirstRowInTbl);
row2Id = F_ApiGetId(docId, row1Id, FP_NextRowInTbl);

/* Add the rows. */
if (row2Id) F_ApiAddRows(docId, row2Id, FV_Below, 2);
. . .
```

Creating commands, menus, and menu item separators

For information on creating and deleting commands, menus, and menu item separators, see Chapter 9, “Manipulating Commands and Menus with the API.”

Deleting objects

To delete objects, use `F_ApiDelete()`.

The syntax for `F_ApiDelete()` is:

```
IntT F_ApiDelete(F_ObjHandleT docId,
                F_ObjHandleT objId);
```

This argument	Means
<code>docId</code>	The ID of the document from which to delete the object
<code>objId</code>	The ID of the object to delete

If `F_ApiDelete()` is successful, it returns `FE_Success`. Otherwise, it returns an error code. There are a number of objects that you can't delete. For example, you can't delete an `FO_Doc` object or an `FO_VarFmt` object that represents a system variable format. For the list of these objects, see "F_ApiDelete()" in the FDK Programmer's Reference guide.

When you delete an object, the API automatically deletes all of that object's child objects. For example, if you delete a frame, the API deletes all the objects in the frame. If you delete an `FO_Tbl` object, the API deletes all the `FO_Row` objects and `FO_Cell` objects in the table. Similarly, if you delete an element in a FrameMaker document, the API deletes all the descendants of that element.

Deleting flows and text frames

When you delete a flow, the API also deletes all the text frames in it (and all the paragraphs in the text frames). If you don't want to delete a text frame when you delete a flow, you must disconnect the text frame from the flow before you delete it. To disconnect a text frame from a flow, set the text frame's `FP_PrevTextFrameInFlow` and `FP_NextTextFrameInFlow` properties to 0.

If you delete a text frame that is not connected to another text frame, the API deletes the flow that contains it.

Deleting table columns and rows

To delete table columns and rows use these functions:

- `F_ApiDeleteCols()` to delete table columns
- `F_ApiDeleteRows()` to delete table rows

The syntax for `F_ApiDeleteCols()` is:

```
IntT F_ApiDeleteCols(F_ObjHandleT docId,
                    F_ObjHandleT tblId,
                    IntT refColNum,
                    IntT numDelCols);
```

This argument	Means
<code>docId</code>	The ID of the document containing the table.
<code>tblId</code>	The ID of the table containing the columns.
<code>refColNum</code>	The first column to delete. Columns are numbered from left to right, starting with column 0.
<code>numDelCols</code>	The number of columns to delete.

`F_ApiDeleteCols()` deletes the column specified by `refColNum` and $(\text{numDelCols} - 1)$ columns to the right of it. If `F_ApiDeleteCols()` is successful, it returns `FE_Success`. Otherwise, it returns an error code. When you delete a table column, the API automatically deletes all the `FO_Cell` objects in the column.

The syntax for `F_ApiDeleteRows()` is:

```
IntT F_ApiDeleteRows(F_ObjHandleT docId,
                    F_ObjHandleT tblId,
                    IntT refRowId,
                    IntT numDelRows);
```

This argument	Means
<code>docId</code>	The ID of the document containing the table
<code>tblId</code>	The ID of the table containing the rows
<code>refRowId</code>	The ID of the first row to delete
<code>numDelRows</code>	The number of rows to delete

`F_ApiDeleteRows()` deletes the row specified by `refRowId` and `(numDelRows-1)` rows below it. If `F_ApiDeleteRows()` is successful, it returns `FE_Success`. Otherwise, it returns an error code. `F_ApiDeleteRows()` deletes only one type of row at a time. If you attempt to delete a range of rows that includes body rows and header or footer rows, `F_ApiDeleteRows()` returns an error. When you delete a table row, the API automatically deletes the `FO_Row` object and all the `FO_Cell` objects in the row.

Implicit property changes

When you create or delete an object, the API automatically updates other objects and properties that are affected. For example, if you delete a paragraph (`FO_Pgf`), the API automatically updates the `FP_NextPgfnFlow` property of the previous paragraph and the `FP_PrevPgfnFlow` property of the next paragraph. Figure 8-4 shows the paragraph objects in a flow before and after an `FO_Pgf` object is deleted.

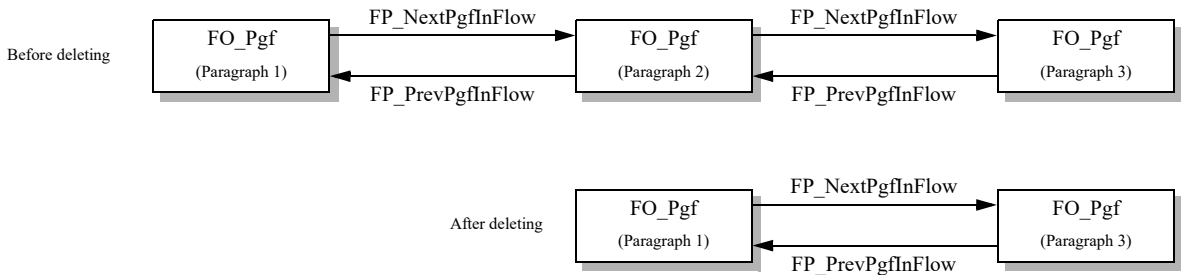


Figure 8-4 *FO_Pgf* objects in a flow before and after deleting an *FO_Pgf* object

Manipulating Commands and Menus with the API

.....

·
·
·
·

This chapter describes Frame API command and menu functionality in detail. For an introduction to using commands and menus in your client’s interface, see “Using commands, menu items, and menus in your client” on page 205.

How the API represents commands and menus

The API uses an `FO_Command` object to represent each command, an `FO_Menu` object to represent each menu, and an `FO_MenuItemSeparator` object to represent each menu item separator in a FrameMaker product session.

Suppose a FrameMaker session has the view-only menu bar shown in Figure 9-1.

File	Edit	Navigation
New...		!fn
Open...		!fo
Quit		!fq

Figure 9-1 *FrameMaker* view-only menu bar

Figure 9-2 shows the objects that represent a FrameMaker product view-only menu bar, the menus it contains, and the items the File menu contains.

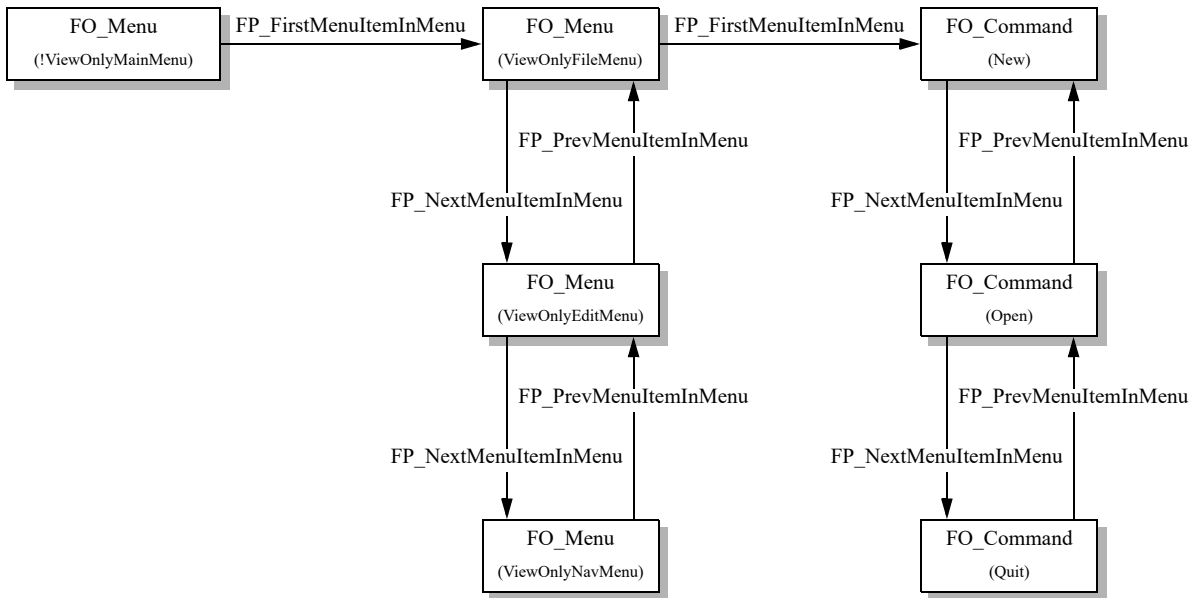


Figure 9-2 *Objects that represent menus and menu items*

.....
IMPORTANT: `FP_FirstMenuItemInMenu`, `FP_PrevMenuItemInMenu`, and `FP_NextMenuItemInMenu` properties can specify menu, menu item, or menu item separator IDs.

The following table lists some of the properties of the `FO_Command` object that represent the Open menu item in Figure 9-1.

Property	Type	Value
<code>FP_CanHaveCheckMark</code>	<code>IntT</code>	<code>False</code>
<code>FP_Fcodes</code>	<code>F_IntsT</code>	<code>{0x310}</code>
<code>FP_KeyboardShortcuts</code>	<code>F_StringsT</code>	<code>{"\\!fo"}</code>
<code>FP_KeyboardShortcutLabel</code>	<code>StringT</code>	<code>"!fo"</code>
<code>FP_Labels</code>	<code>F_StringsT</code>	<code>{"Open..."}</code>
<code>FP_MenuItemIsEnabled</code>	<code>IntT</code>	<code>True</code>
<code>FP_Name</code>	<code>StringT</code>	<code>"Open"</code>

You can get and set `FO_Command`, `FO_Menu`, and `FO_MenuItemSeparator` properties the same way you get and set the properties of other API objects. For more information on getting and setting object properties, see Chapter 5, “Getting and Setting Properties.”

Lists of menus, menu items, and commands in a session

The API maintains a linked list of all the menus and menu items in a session. The `FO_Session` property, `FP_FirstMenuItemInSession`, specifies the ID of the first object in the list. The menu or menu item property, `FP_NextMenuItemInSession`, specifies the next object in the list. The list is not in any particular order.

The API also maintains a linked list of all the commands in a session. The `FO_Session` property, `FP_FirstCommandInSession`, specifies the ID of the first command in the list. The `FO_Command` property, `FP_NextCommandInSession`, specifies the next command in the list. Like the list of menus and menu items, this list is not in any particular order.

For a complete listing of all the available menus in the FrameMaker product see, “Getting the IDs of FrameMaker product menus and menu bars” on page 205.

Getting and setting menu and menu item properties

A single menu or command can have multiple instances in a FrameMaker product session; each FrameMaker product menu can contain an instance. . The API uses only one object to represent all instances of a menu, command, or menu item separator.

Most properties of an `FO_Menu`, `FO_Command`, or `FO_MenuItemSeparator` object apply to all instances of the object. For example, if you use the following call to set the label of the Cut command:

```
. . .
F_ObjHandleT cutCmdId;
cutCmdId = F_ApiGetNamedObject(FV_SessionId, FO_Command, "Cut");
F_ApiSetString(FV_SessionId, cutCmdId, FP_Label, "Excise");
. . .
```

the API changes the label of all instances of the Cut command to `Excise`.

The following properties apply only to individual instances of an `FO_Menu`, `FO_Command`, or `FO_MenuItemSeparator` object:

- `FP_PrevMenuItemInMenu`
- `FP_NextMenuItemInMenu`

When you get or set these properties, you must indicate which instance of the object you want to get or set them for. To do this, set the first parameter of the `F_ApiGetId()` or `F_ApiSetId()` function to the ID of the menu containing the instance.

For example, the following code gets the ID of the menu item above Cut on the Edit menu. Then it gets the ID of the menu item above Cut on the document window pop-up menu.

```
. . .
F_ObjHandleT editMenuId, cutCmdId, docPopupMenuId,
              prevItemOnEditMenuId, prevItemOnDocMenuId;

editMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
                                "EditMenu");
docPopupMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
                                     "!MakerPopup");
cutCmdId = F_ApiGetNamedObject(FV_SessionId, FO_Command, "Cut");
prevItemOnEditMenuId = F_ApiGetId(editMenuId, cutCmdId,
                                  FP_PrevMenuItemInMenu);
prevItemOnDocMenuId = F_ApiGetId(docPopupMenuId, cutCmdId,
                                 FP_PrevMenuItemInMenu);
. . .
```

Getting the IDs of commands and menus

To manipulate a command or menu, you need its ID. If you know its name, the simplest way to get its ID is to call `F_ApiGetNamedObject()`. For example, the following code gets the IDs of the FrameMaker main menu bar and the File menu:

```
. . .  
F_ObjHandleT fileMenuId, mainMenuBarId;  
mainMenuBarId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,  
                                     "!MakerMainMenu");  
fileMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,  
                                  "FileMenu");
```

```
. . .
```

If a menu or command with the specified name doesn't exist, `F_ApiGetNamedObject()` returns 0.

If you don't know a command's name, there are several lists of objects you can traverse to get its ID:

- If you know the command is on a particular menu, traverse the list of menu items on the menu.
- If you know the command is on a menu but you don't know which menu, traverse the list of menu items in the session.
- If the command isn't on any menu, traverse the list of commands in the FrameMaker product session.

Because there are so many FrameMaker product commands, traversing all of them can be slow. If a command is a menu item, get its ID by traversing the list of menu items on a menu or in a session instead of traversing the list of commands in the session.

For example, if you don't know a menu item's name, but you know that its label is Database, you can use the following code to get its ID:

```
. . .  
F_ObjHandleT itemId;  
StringT itemName;  
  
itemId = F_ApiGetId(0, FV_SessionId, FP_FirstMenuItemInSession);  
while(itemId)  
{  
    itemName = F_ApiGetString(FV_SessionId, itemId, FP_Label);  
    if(F_StrEqual(itemName, "Database")) break;  
    itemId = F_ApiGetId(FV_SessionId, itemId,  
                        FP_NextMenuItemInSession);  
    F_Free(itemName);  
}  
. . .
```

Determining a session's menu configuration

The `F_ApiGetNamedObject()` function indicates only whether a command or menu exists. It does not indicate whether it appears on a menu. To determine whether a command or menu appears on a specific menu, call `F_ApiMenuItemInMenu()`.

The syntax for `F_ApiMenuItemInMenu()` is:

```
F_ObjHandleT F_ApiMenuItemInMenu (F_ObjHandleT menuId,  
    F_ObjHandleT menuItemId,  
    BoolT recursive);
```

This argument	Means
<code>menuId</code>	The menu or menu bar to search.
<code>menuItemId</code>	The ID of the menu item or menu to search for.
<code>recursive</code>	Flag specifying whether to search the submenus on the menu specified by <code>menuId</code> . Specify <code>True</code> to search them.

`F_ApiMenuItemInMenu()` returns the ID of the menu on which it finds the specified menu or menu item. If it finds the menu or menu item on a submenu of the menu specified by `menuId`, it returns the ID of the submenu.

For example, the following code determines whether the Copy menu item is on the Edit menu or any of its submenus:

```
. . .  
F_ObjHandleT copyCmdId, editMenuId, copyMenuId;  
  
editMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,  
    "EditMenu");  
copyCmdId = F_ApiGetNamedObject(FV_SessionId, FO_Command,  
    "Copy");  
copyMenuId = F_ApiMenuItemInMenu(editMenuId, copyCmdId, True);  
  
if(copyMenuId == editMenuId)  
    F_Printf(NULL, "Copy is on the Edit menu.");  
else if (copyMenuId)  
    F_Printf(NULL, "Copy is on a submenu of the Edit Menu.");  
else  
    F_Printf(NULL, "Copy is not on the Edit menu.");  
. . .
```

If you want to enumerate the items at all hierarchical levels of a menu, keep in mind that a menu item's `FP_PrevMenuItemInMenu` and `FP_NextMenuItemInMenu` properties specify menu items or menus only at the same hierarchical level. To list all the menu items on a menu, you must recursively traverse each of its submenus. For example, the following function prints the IDs of all the items on a specified menu and its submenus:

```

. . .
VoidT printMenuItemIds(menuId)
    F_ObjHandleT menuId;
{
    F_ObjHandleT itemId;

    itemId = F_ApiGetId(FV_SessionId, menuId,
                       FP_FirstMenuItemInMenu);
    while(itemId)
    {
        F_Printf(NULL, "Item ID: 0x%x\n", itemId);
        if(F_ApiGetObjectTypeId(menuId, itemId) == FO_Menu)
            printMenuItemIds(itemId); /* Recursive call */
        itemId = F_ApiGetId(menuId, itemId,
                             FP_NextMenuItemInMenu);
    }
}
. . .

```

Arranging menus and menu items

The API allows you to add a command to multiple menus and to reorder and delete menus and menu items. The following sections describe this functionality in detail.

Adding a command to multiple menus

You can't use the `F_ApiDefineAndAddCommand()` function discussed in "Defining commands and adding them to menus" on page 207 to add a command to multiple menus. Instead, you must use `F_ApiDefineCommand()` to create the command and `F_ApiAddCommandToMenu()` to add it to the menus.

The syntax for `F_ApiDefineCommand()` is:

```
F_ObjHandleT F_ApiDefineCommand(IntT cmd,
    StringT tag,
    StringT label,
    StringT shortcut);
```

This argument	Means
<code>cmd</code>	The integer that the FrameMaker product passes to your client's <code>F_ApiCommand()</code> function when the user chooses the menu item or types the keyboard shortcut for the command.
<code>tag</code>	A unique name to identify the command.
<code>label</code>	The title of the command as it appears on the menu.
<code>shortcut</code>	The keyboard shortcut sequence.

The syntax for `F_ApiAddCommandToMenu()` is:

```
IntT F_ApiAddCommandToMenu(F_ObjHandleT toMenuId,
    F_ObjHandleT cmdId);
```

This argument	Means
<code>toMenuId</code>	The menu to which to add the command
<code>cmdId</code>	The ID of the command

For example, the following code creates a command named Grammar and adds it to the Edit and Utilities menus:

```
. . .
#define GRAMMAR_CMD 1
F_ObjHandleT editMenuId, utilsMenuId, grammarCmdId;
editMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
    "EditMenu");
utilsMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
    "UtilitiesMenu");
grammarCmdId = F_ApiDefineCommand(GRAMMAR_CMD, "Grammar",
    "Grammar...", "\\!GG");
F_ApiAddCommandToMenu(editMenuId, grammarCmdId);
F_ApiAddCommandToMenu(utilsMenuId, grammarCmdId);
. . .
```

You can also use `F_ApiAddCommandToMenu()` to add FrameMaker-defined commands to multiple menus. For example, the following code adds the Compare Documents command to the Edit and Utilities menus:

```
. . .
F_ObjHandleT editMenuId, compareCmdId, UtilsMenuId;
compareCmdId = F_ApiGetNamedObject(FV_SessionId, FO_Command,
    "DocCompare");
editMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
    "EditMenu");
UtilsMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
    "UtilitiesMenu");
F_ApiAddCommandToMenu(editMenuId, compareCmdId);
F_ApiAddCommandToMenu(UtilsMenuId, compareCmdId);
. . .
```

A user's menu configuration file can also add a FrameMaker or a client command to several menus. For example, the following lines of a menu configuration file add a client-defined command named Grammar to the Edit and Utilities menus.

```
<Command Grammar>
<Add Grammar <Menu EditMenu>>
<Add Grammar <Menu UtilitiesMenu>>
```

Removing menus and menu items

To remove a menu or menu item, call `F_ApiDelete()` with the first parameter set to the ID of the menu that contains the menu or menu item and the second parameter set to the ID of the menu or menu item. `F_ApiDelete()` deletes a menu or menu item from only the menu you specify. If a menu or menu item is on several menus, you must make a separate `F_ApiDelete()` call to remove it from each menu.

For example, the following code removes the Copy command from the Edit menu:

```
. . .
F_ObjHandleT copyCmdId, editMenuId;
editMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
    "EditMenu");
copyCmdId = F_ApiGetNamedObject(FV_SessionId, FO_Command,
    "Copy");
F_ApiDelete(editMenuId, copyCmdId);
. . .
```

Reordering menus and menu items

To change a menu or menu item's position on a menu, set its `FP_NextMenuItemInMenu` or `FP_PrevMenuItemInMenu` properties to specify the IDs of other menus or menu items on the menu. You need to set only one of these properties. FrameMaker automatically sets the other one for you.

For example, if the Cut and Copy commands are on the Edit menu, you can use the following code to make Copy appear above Cut:

```
. . .
F_ObjHandleT cutCmdId, copyCmdId, editMenuId;
editMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
                                "EditMenu");
cutCmdId = F_ApiGetNamedObject(FV_SessionId, FO_Command,
                               "Cut");
copyCmdId = F_ApiGetNamedObject(FV_SessionId, FO_Command,
                                "Copy");
F_ApiSetId(editMenuId, copyCmdId, FP_NextMenuItemInMenu,
           cutCmdId);
. . .
```

The following `F_ApiSetId()` call has the same effect as the call in the code above:

```
F_ApiSetId(editMenuId, cutCmdId, FP_PrevMenuItemInMenu,
           copyCmdId);
```

To move a menu or menu item to the top of a menu, set its `FP_PrevMenuItemInMenu` property to 0. To move it to the bottom of a menu, set its `FP_NextMenuItemInMenu` property to 0. For example, the following code moves the Copy menu item to the top of the Edit menu:

```
. . .
F_ObjHandleT copyCmdId, editMenuId;
editMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
                                "EditMenu");
copyCmdId = F_ApiGetNamedObject(FV_SessionId, FO_Command,
                                "Copy");
F_ApiSetId(editMenuId, copyCmdId, FP_PrevMenuItemInMenu, 0);
. . .
```

If a menu is on a menu bar, setting its `FP_PrevMenuItemInMenu` property to 0 moves it to the leftmost position on the menu bar; setting its `FP_NextMenuItemInMenu` property to 0 moves it to the rightmost position.

You can't move a menu or menu item to another menu by setting its `FP_NextMenuItemInMenu` or `FP_PrevMenuItemInMenu` properties. Instead, you must delete it and then add it to the menu on which you want it to appear. For example, the following code moves the Font menu from the Format menu to the main menu bar:

```
. . .
F_ObjHandleT formatMenuId, fontMenuId, mainMenuBarId;
. . .
/* Delete instance of Font menu on the Format menu. */
F_ApiDelete(formatMenuId, fontMenuId);
F_ApiAddMenuToMenu(mainMenuBarId, fontMenuId);
. . .
```

Changing the menu set

The user can switch to a menu set by choosing the menu set from View>Menus. Your client can switch menu sets programmatically by setting the session's `FP_CurrentMenuSet` property. For example, the following code switches to quick menus:

```
. . .
F_ApiSetInt(0, FV_SessionId, FP_CurrentMenuSet, FV_MENU_QUICK);
. . .
```

You can't switch to custom menus unless you have already loaded a custom menu file. To load a custom menu file, call `F_ApiLoadMenuCustomizationFile()`.

Manipulating menu item separators

You can manipulate menu item separators (FO_MenuItemSeparator objects) the same way you manipulate menus and menu items, with the following limitations:

- If there is more than one separator on a menu, each separator must have a different name.
- A separator can't be the first or the last item on a menu.
- Separators can't appear next to each other.

FrameMaker provides six predefined separators: !Separator, Separator1, Separator2, Separator3, Separator4, and Separator5. To get the ID of one of these separators, use F_ApiGetNamedObject () as follows:

```
. . .  
F_ObjHandleT separatorId;  
separatorId = F_ApiGetNamedObject (FV_SessionId,  
                                   FO_MenuItemSeparator, "Separator1");  
. . .
```

You can use any predefined separator name when you add a separator to a menu. For example, you could use the predefined separator name Separator5 for the first separator you add to a menu. However, you should try to use the predefined separator name that corresponds to a separator's position among the separators on a menu. For example, the first separator on a menu should use the predefined separator name Separator1 and the second separator should use the predefined separator name Separator2. This makes it easier for other clients and users who modify menu configuration files to manipulate the separators.

You can use the same separator on different menus. For example, if you create two menus that contain two separators, you can use Separator1 and Separator2 on each menu.

Adding, moving, and deleting separators

You can add, move, and delete separators the same way you add, move, and delete commands and menu items. For example, the following code removes the separator that appears after the Conditional Text menu item on the Special menu. It then adds a separator and moves it above the Conditional Text menu item.

```

. . .
F_ObjHandleT specialMenuId, separatorId, conditionCmdId;

specialMenuId = F_ApiGetNamedObject (FV_SessionId,
                                     FO_Menu, "SpecialMenu");
separatorId = F_ApiGetNamedObject (FV_SessionId,
                                   FO_MenuItemSeparator, "Separator1");

conditionCmdId = F_ApiGetNamedObject (FV_SessionId,
                                      FO_Command, "ConditionalText");

/* Delete the separator. */
F_ApiDelete(specialMenuId, separatorId);

/* Add it back and move it below the Conditional Text item. */
F_ApiAddCommandToMenu(specialMenuId, separatorId);
F_ApiSetId(specialMenuId, separatorId,
           FP_NextMenuItemInMenu, conditionCmdId);
. . .

```

Creating separator objects

Because separators appear the same and you can use the same separator on multiple menus, you will normally need only the predefined separators. If you need additional separators, you can create them with `F_ApiNewNamedObject()` as follows:

```

. . .
F_ObjHandleT separatorId;
separatorId = F_ApiNewNamedObject (FV_SessionId,
                                   FO_MenuItemSeparator, "MySeparator");
. . .

```

Getting and setting menu item labels

Most FrameMaker product menu items have only one label. For example, the label of the Cut command is always `Cut`. However, some menu items have different labels for different contexts. For example, the label of the TableConvert command is `Convert to Table` when paragraph text is selected, but `Convert to Paragraphs` when the insertion point is in a table or table cells are selected.

The `FP_Labels` property specifies the labels a menu item can have in different contexts. If a menu item has one label for all contexts, its `FP_Labels` property specifies only that label. Otherwise, its `FP_Labels` property specifies pairs of strings with the following format:

```
context, label,
```

where *context* specifies a context and *label* specifies the menu item label that appears when that context is applicable. The following table lists some of the values *context* can have.

Context value	When the label is displayed
Book	When a book is active
Document	When a document is active
Long	When a menu item is on a pull-down menu or the document pop-up menu
ToTable	When text that is not a table or text line is selected
ToText	When the insertion point is in a table cell or one or more table cells are selected
Short	When a menu item is on a pull-right menu or one of the formatting bar menus

For example, the strings specified by the `FP_Labels` property of the TableConvert command are:

```
{"ToTable", "Convert to Table..."},
  {"ToText", "Convert to Paragraphs..."}
```

Setting the labels of FrameMaker product menu items

You can change the labels of FrameMaker product menu items. If a FrameMaker product menu item has labels for different contexts, you can change only the strings that specify the labels. You can't change the strings that specify the contexts in which the labels appear. For example, the following code changes the labels for the TableConvert command:

```
. . .
#include "fstrings.h"

F_ObjHandleT cmdId;
StringT labels[4];
F_StringsT myLabels;

. . .
labels[0] = (StringT) F_StrCopyString("ToTable");
labels[1] = (StringT) F_StrCopyString("Make table out of this");
labels[2] = (StringT) F_StrCopyString("ToText");
labels[3] = (StringT) F_StrCopyString("Convert to paragraphs");
myLabels.len = 4;
myLabels.val = (StringT *)labels;
F_ApiSetStrings(FV_SessionId, cmdId, FP_Labels, &myLabels);

. . .
```

Setting the labels of client-defined menu items

A client-defined menu item can have only one label for all contexts. Its `FP_Labels` property should specify only one string. For example, the following code sets the label of a client-defined menu item to `My Item`:

```
. . .
F_StringsT myLabels;
F_ObjHandleT cmdId;
StringT labels[1];

labels[0] = (StringT) "My Item";
myLabels.len = 1;
myLabels.val = (StringT *)labels;
F_ApiSetStrings(FV_SessionId, cmdId, FP_Labels, &myLabels);

. . .
```


Manipulating expandomatic menu items

An *expandomatic* menu item is a dynamically determined set of menu items that FrameMaker products treat as a single menu item. For example, the list of paragraph formats that appears on the lower part of the Format>Paragraphs menu is an expandomatic menu item named !ShowParagraphTags. FrameMaker products currently use the following expandomatic menu items:

Expandomatic menu item name	Description
!ShowCharacterTags	The list of character formats available in the current document
!ShowDocumentWindows	The list of document windows in the current session
!ShowFilesRecentlyVisited	The list of the last five files opened
!ShowFontChoices	The list of font families available in the session
!ShowImportFiles	The list of open files that a user can import into the current document
!ShowParagraphTags	The list of paragraph formats available in the current document

A FrameMaker product can change the contents of an expandomatic menu item during a session. For example, when the user sets the insertion point in a document, the FrameMaker product changes the !ShowParagraphTags menu item to list the paragraph formats available in the document. If the user adds or deletes a paragraph format, the FrameMaker product updates the list to reflect the change.

You can manipulate an expandomatic menu item just as you manipulate any other menu item. However, you can't manipulate the individual items that constitute the expandomatic item. For example, you can move or remove the entire !ShowParagraphTags item, but you can't move or remove an individual item, such as Body, that appears on it.

You can get the ID of an expandomatic item with the code similar to the following:

```
. . .
F_ObjHandleT itemId;
itemId = F_ApiGetNamedObject(FV_SessionId, FO_Command,
                             " !ShowParagraphTags");
. . .
```

Individual items in an expandomatic item don't have IDs. You can determine which items an expandomatic menu item contains by getting object properties. For example,

you can determine which items `!ShowFontChoices` contains by getting the session property `FP_FontFamilyNames`.

Using check marks

FrameMaker products display check marks next to some menu items to indicate the state of a setting or option. For example, when borders are visible in a document, a FrameMaker product displays a check mark next to the Borders menu item. Your client can also display check marks next to its menu items. Menu items have two properties that control check marks:

- `FP_CanHaveCheckMark`, which specifies whether an item can have a check mark
- `FP_CheckMarkIsOn`, which specifies whether a check mark appears next to an item

To use a check mark with a menu item, set `FP_CanHaveCheckMark` to `True`. Then make the check mark visible by setting `FP_CheckMarkIsOn` to `True`, or invisible by setting it to `False`.

.....
IMPORTANT: *Once you set `FP_CanHaveCheckMark` to `True`, resetting it to `False` has no effect. Setting `FP_CheckMarkIsOn` has an effect only when `FP_CanHaveCheckMark` is set to `True`.*

Using context-sensitive commands and menu items

Many FrameMaker product commands and menu items change depending on the context. For example, the Group command is disabled when there are no objects selected. The API provides properties that allow you to make your client's commands and menu items context sensitive like FrameMaker product commands. The following sections describe how to use these properties.

Enabling commands for specific contexts

The `FP_EnabledWhen` property specifies the contexts in which a command is enabled. The following table lists some of the values `FP_EnabledWhen` can have.

<code>FP_EnabledWhen</code> value	Context in which a menu item is active
<code>FV_ENABLE_ALWAYS_ENABLE</code>	All contexts. This is the default value.
<code>FV_ENABLE_ALWAYS_DISABLE</code>	No context. Setting <code>FP_EnabledWhen</code> to this value completely disables the menu item.
<code>FV_ENABLE_IS_TEXT_SEL</code>	Text is selected in a paragraph or a graphic text line.
<code>FV_ENABLE_IN_TEXT_LINE</code>	The insertion point or selection is in a graphic text line.

For a complete list of the values `FP_EnabledWhen` can have, see “FO_Command” in the FDK Programmer’s Reference guide.

When you create a command, `FP_EnabledWhen` has the default value, `FV_ENABLE_ALWAYS_ENABLE`. To completely disable a command, set its `FP_EnabledWhen` property to `FV_ENABLE_ALWAYS_DISABLE`. To reenable a command, reset `FP_EnabledWhen` to `FV_ENABLE_ALWAYS_ENABLE`. To enable a command only in a specific context, set its `FP_EnabledWhen` property to one of the other listed values.

For example, the following code creates a command that is enabled only when text is selected:

```
. . .
F_ObjHandleT cmdId;

cmdId = F_ApiDefineCommand(1, "Transpose", "Transpose Words", "");
F_ApiSetInt(FV_SessionId, cmdId, FP_EnabledWhen,
            FV_ENABLE_IS_TEXT_SEL);
. . .
```

If a command is a menu item, it appears dimmed when it is disabled. You can determine whether a menu item is disabled by getting its `FP_MenuItemIsEnabled` property. This is easier than getting its `FP_EnabledWhen` property and determining whether the specified context currently applies. You can’t set the `FP_MenuItemIsEnabled` property.

Using shift menu items

FrameMaker products provide several *shift menu items*, menu items that change when the user holds down the Shift key. For example, when the user holds down the Shift key and displays the File menu, the label of the Close menu is `Close All Open Files` instead of `Close`. If the user chooses the menu item, the FrameMaker product closes all open files.

Shift menu items actually represent two separate commands, which are linked by their `FP_HasShiftOrUnshiftCommand` and `FP_ShiftOrUnshiftCommand` properties. For example, the Close menu item represents the commands Close and CloseAll. The following table shows the values of their `FP_HasShiftOrUnshiftCommand` and `FP_ShiftOrUnshiftCommand` properties:

Command	Property	Value
Close	<code>FP_HasShiftOrUnshiftCommand</code>	<code>FV_ITEM_HAS_SHIFT_COMMAND</code>
	<code>FP_ShiftOrUnshiftCommand</code>	ID of CloseAll command
CloseAll	<code>FP_HasShiftOrUnshiftCommand</code>	<code>FV_ITEM_HAS_UNSHIFT_COMMAND</code>
	<code>FP_ShiftOrUnshiftCommand</code>	ID of Close command

The API allows clients to create and use shift menu items. To combine two commands into a shift menu item, you have to set the `FP_HasShiftOrUnshiftCommand` and `FP_ShiftOrUnshiftCommand` properties for only one of the commands. The API automatically sets the properties of the other command for you.

For example, the following code creates a shift menu item representing the client-defined commands, Checkin and CheckinAll. If the user displays the Edit menu normally, the label Check in File appears on it. If the user holds down the Shift key and displays the Edit menu, the label Check in All Open Files appears on it.

```

. . .
#define CHECKIN_CMD 1
#define CHECKIN_ALL_CMD 2
F_ObjHandleT editMenuId, checkinCmdId, checkinAllCmdId;
editMenuId = F_ApiGetNamedObject(FV_SessionId, FO_Menu,
                                "EditMenu");
checkinCmdId = F_ApiDefineAndAddCommand(CHECKIN_CMD, editMenuId,
                                        "Checkin", "Check in File", "");
checkinAllCmdId = F_ApiDefineCommand(CHECKIN_ALL_CMD,
                                     "CheckinAll", "Check in All Open Files", "");

F_ApiSetInt(editMenuId, checkinCmdId,
            FP_HasShiftOrUnshiftCommand, FV_ITEM_HAS_SHIFT_COMMAND);
F_ApiSetId(editMenuId, checkinCmdId, FP_ShiftOrUnshiftCommand,
           checkinAllCmdId);
. . .

```


Creating Custom Dialog Boxes for Your Client

.....

10

⋮

This chapter describes how to use FrameMaker products to create and modify custom dialog boxes that you can use in your client’s user interface. You can create a dialog box that contains the following items:

- Boxes
- Buttons
- Checkboxes
- Pop-up menus (with a standard appearance or drawn from bitmap images)
- Radio buttons
- Scroll bars
- Scroll lists
- Text boxes (with one or more lines)

If your client’s user interface requires only simple modal dialog boxes, you may not need to create custom dialog boxes. The API provides several simple, ready-made modal dialog boxes. For information on using these dialog boxes, see “Using API dialog boxes to prompt the user for input” on page 195.

Overview

The process of including a custom dialog box in your client involves the following general steps:

1 Create the custom dialog box.

Creating a dialog box involves designing its layout and items and saving this information in a file format that can be used to build your FDK client. Instructions for this step are presented in this chapter.

2 Write the code in your client to manipulate the dialog box.

After you create a custom dialog box for your client, you need to add code to your client to manipulate it. For more information, see Chapter 11, “Handling Custom Dialog Box Events.”

3 Compile the dialog box with your client in the build process.

After you write the code for your client, you can compile the code with the files containing dialog box information.

This overview section describes the fundamentals behind the first step of this process, creating dialog boxes for clients. The later sections of this chapter describe the specific procedures in this step of the process.

The end of this chapter lists some helpful tips to keep in mind when editing dialog boxes.

Designing a dialog box in a FrameMaker product

You can use a FrameMaker product as a dialog editing application. FrameMaker products can represent dialog box information as special Frame graphic objects. You can then modify the dialog box and its items just as you would modify standard Frame graphic objects.

Figure 10-1 shows a dialog box edited within a FrameMaker product.

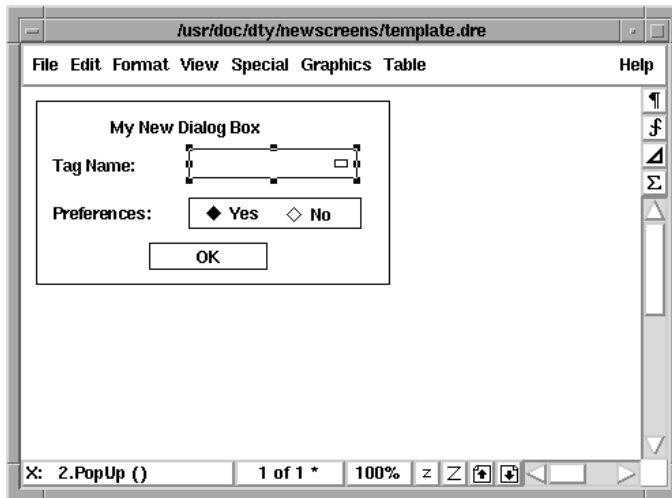


Figure 10-1 Using a FrameMaker product to design a dialog box

To edit dialog box information through a FrameMaker product, you open a special type of file called a *Frame dialog resource (DRE)* file. As Figure 10-2 shows, when you open a DRE file in a FrameMaker product, the FrameMaker product translates the dialog box information into a graphic representation of the dialog box.

This is similar to opening a Frame binary document in a FrameMaker product. When you open a Frame binary document, the FrameMaker product translates the document information into a graphic representation of the document.

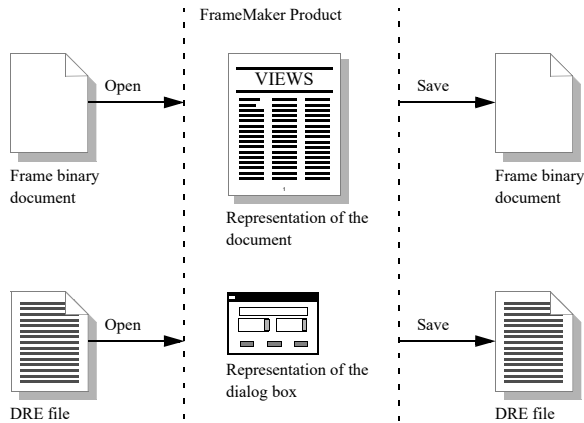


Figure 10-2 Comparison between opening a Frame binary file and a DRE file

The rest of this section describes Frame DRE files and how FrameMaker products interpret these files.

Frame DRE files

A DRE file is a text file that uses special syntax to describe a dialog box and its items. The following lines from a DRE file illustrate how the DRE file syntax describes the OK button in a dialog box:

```
<Button
  <MBaseLine 10 327 92>
  <WBaseLine 59 231 53>
  <XBaseLine 63 368 64>
  <Label OK>
  <Active No>
  <HypertextHelp clnthelp:button>
> # 29
```

FrameMaker products recognize this syntax and interpret the descriptions of the dialog box and its items as special Frame graphic objects. Figure 10-3 shows the object properties for the OK button described in the previous example.

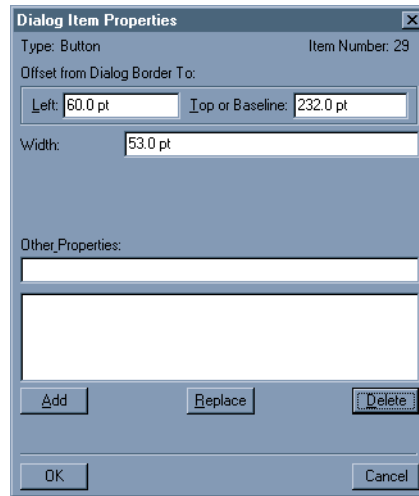


Figure 10-3 Object properties for a button in a dialog box

By moving and resizing these special objects and changing their properties within a FrameMaker product, you modify the dialog description in the DRE file. When you save a DRE file, the changes that you make are saved back to the file in the special DRE syntax.

Saving dialog box information

When you save a Frame DRE file, all the dialog box information is saved in text format in the special DRE syntax. FrameMaker writes out a Windows dialog resource file (.dlg) and an extra dialog information file (.xdi). The files use the same base name as the DRE file (for example, if your DRE file is named `mydialog.dre`, the FrameMaker product writes out the additional files `mydialog.dlg` and `mydialog.xdi`). These files are resource description files recognized by Windows and are used to compile the dialog box resources with your FDK client.

The rest of this section briefly describes how dialog boxes are included in your FDK client.

Dialog box information

In the Windows build process, dialog box information needs to be provided in a Windows dialog resource file (a `.dlg` file). Additional information specific to FrameMaker dialog boxes needs to be provided in a separate file (an `.xdi` file).

For this reason, when you save a DRE file in a FrameMaker product, the FrameMaker product also writes out the same dialog box information in a `.dlg` file and an `.xdi` file.

Figure 10-4 shows the process of creating a dialog box for a client.

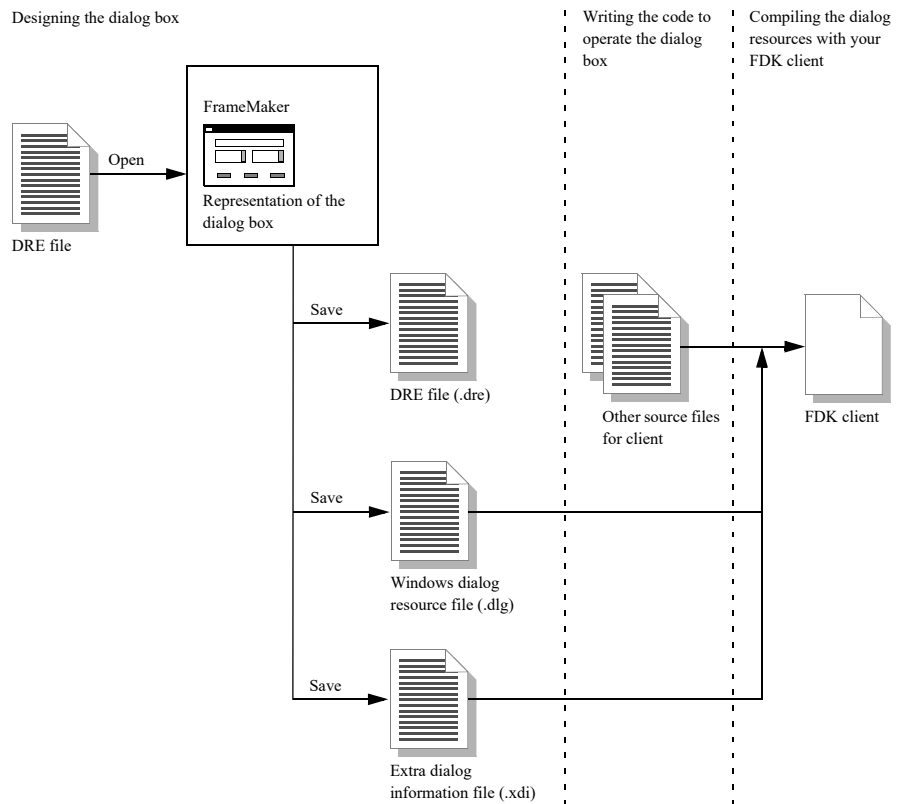


Figure 10-4 Creating a dialog box for a client

Invoking the dialog box by name

When you write the code to invoke the dialog box, use the name of the resource as an argument for opening the dialog resource. The dialog resource is named after the DRE file. For example, if your DRE file is named `mydialog.dre`, the build process creates a dialog resource named `mydialog`.

For more details on displaying dialog boxes, see “Opening dialog resources” on page 448.

How to create a dialog box

The process of creating a dialog box involves the following general steps. Each step is described in more detail in subsequent sections of this chapter.

- 1 *Using a FrameMaker product, create a new DRE file from an existing DRE file.*

For details, see the next section.

- 2 *Design the layout of the dialog box.*

For more information, see the section “Designing the layout of the dialog box” on page 415.

- 3 *Specify the properties of the dialog box.*

For more information, see the section “Setting the properties of the dialog box” on page 419.

- 4 *Specify the properties of the dialog items.*

For details, see the section “Setting the properties of a dialog item” on page 423.

- 5 *Save the new DRE file.*

On some platforms, this creates platform-specific resource files. For details, see the section “Saving a DRE file” on page 431.

- 6 *Test the dialog box.*

For more information, see the section “Testing a dialog box” on page 433.

Creating a DRE file

The first step in creating a dialog box is to create a DRE file. This file stores all the information about a dialog box and its items.

Since dialog boxes and dialog items are different objects than standard Frame graphic objects, you cannot create a new file (such as a blank portrait document, for example) and draw the dialog box and its items. You must start from an existing DRE file that already contains these objects.

Also, although you may find that you are able to copy and paste dialog items from a DRE file into a regular Frame document, you should not use a Frame document to create a dialog box. A saved Frame document cannot be converted to the dialog resources necessary to create a dialog box.

To create a new DRE file, start a FrameMaker product and use the DRE file as a template, as follows:

1 *Choose New from the File menu in the FrameMaker product.*

Using the New command ensures that you don't overwrite the template.

2 *Select an existing DRE file.*

You can use the DRE template file provided with the FDK. You can find the template file in the following locations:

- `fdk_install_dir\samples\dre\template.dre`

where `fdk_install_dir` is the directory in which you installed the FDK.

If you have already created your own DRE files, you can select one of them.

3 *Click Create.*

The FrameMaker product displays the DRE file in a standard Frame document window.

The dialog box and its items appear on a single page. The dialog box is displayed as a rectangle, defining the boundaries of the dialog box.

Each item in the dialog box is displayed as a special Frame graphic object. Although these objects appear to be composed of standard Frame graphic objects (for example, a checkbox is displayed as a rectangle and a text line), you cannot ungroup a dialog item into separate objects.

A DRE file can describe a number of different types of dialog items, including:

- Boxes
- Buttons
- Checkboxes
- Radio buttons
- Text boxes
- Multiline text boxes
- Pop-up menus
- Image pop-up menus (pop-up menus that appear as bitmap images)
- Labels
- Scroll bars
- Scroll lists

When you select an item, the document window status bar displays information about the selected item in the following format:

```
view: item_number.item_type (related_item_number)
```

where

- *view* is a letter specifying the platform view of the dialog box, which is W for Windows

- *item_number* is the number of the selected item.

In DRE files, the items in a dialog box are identified by unique numbers. Item numbers start from 0 and increase sequentially. You use these numbers in your client code to identify items in the dialog box.

- *item_type* is the item's type.
- *related_item_number* specifies the number of a related item.

Some types of items (such as text boxes, image pop-up menus, and radio buttons) can be related to other items. If there is no related item, the status bar displays an empty pair of parentheses.

If the item type does not support related items (such as buttons or labels), the status bar does not display parentheses.

For information about related items, see “Relating items in a dialog box” on page 430.

If you select more than one item, the status bar displays information on the item with the lowest number.

Figure 10-5 shows the status bar when an item is selected.

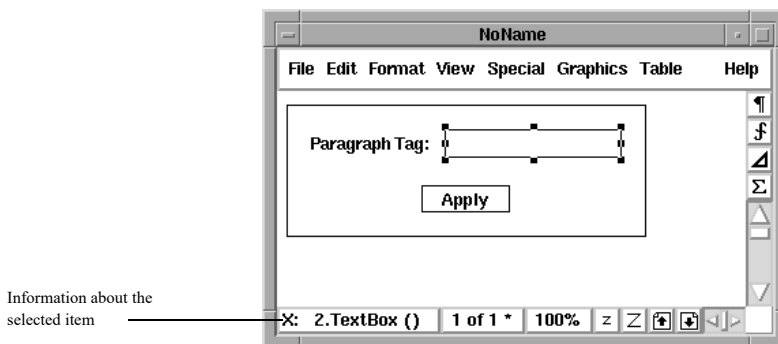


Figure 10-5 Item information in the status bar

Designing the layout of the dialog box

FrameMaker products display the dialog box information graphically in a standard Frame document window. Within a FrameMaker product, you can:

- Manipulate the dialog box and its items as graphic objects
- Add new items
- Delete existing items

The rest of this section describes the specific steps for each of these tasks.

Manipulating the dialog box and its items

When designing the layout of the dialog box, you can manipulate the dialog box and its items in the same way that you manipulate standard Frame graphic objects.

Moving and resizing the dialog box and its items

To move the dialog box, select it and drag it to its new position. To resize the dialog box, select it and drag one of its object handles.

Do not manually resize the height of a multiline text box or a scroll list. Multiline text boxes and scroll lists have special properties that define height in terms of the number of lines of text displayed. To resize a multiline text box or scroll list, set the height of the item in the Object Properties dialog box for the item. For details, see “Setting basic dialog item properties” on page 424.

Also, the length of a label, check box, or radio button is defined by the length of the text associated with the item. To change the length of any of these items, specify a shorter or longer string of text for the item.

Using commands to modify the dialog box and its items

You can use the following commands from the Graphics menu when working with a dialog box and its items:

- Group
- Ungroup (only on items you've manually grouped)
- Bring to Front
- Send to Back
- Align
- Distribute
- Scale (see the previous caveats about resizing)
- Object Properties
- Snap

Note that you cannot ungroup a single dialog item into smaller components. For example, you cannot ungroup a checkbox into a box and a label.

For example, you can use the Align command to align dialog box items. When aligning the bottoms of dialog items, note that the bottom of any item containing text is defined by the baseline of its text line.

Some commands affect the display of graphic objects in a DRE file but have no effect on the actual dialog box that you create from the file.

- You can use the Group command to group dialog items for convenience while laying out the dialog box. Although the grouping works when you are modifying the file, the grouping information is not saved in the DRE file. In other words, grouping has no effect on the appearance or functionality in the actual dialog box. For example, if you group checkboxes together, the grouping does not affect the way the checkboxes work.
- Although you can change the line, fill, and color properties of graphic objects in the DRE file, this does not change the appearance of items in the actual dialog box.
- The Bring to Front and Send to Back commands do not change the appearance of items in the actual dialog box. Although you can use them while editing the DRE file, you should not allow dialog items to overlap.

The following commands from the Graphics menu have no effect on a dialog box and its items:

- Reshape
- Smooth

- Unsmooth
- Flip Up/Down
- Flip Left/Right
- Rotate
- Set Number of Sides

.....
IMPORTANT: *The Undo command discards only changes in size and position. The Undo command does not discard any other changes.*
.....

Redisplaying the dialog box

Measurements in DRE files must use whole number values. If you group items, align items, or work in a zoomed DRE file, the FrameMaker product might not use whole numbers for measurements. As a result, the display of the dialog box might differ from the actual measurements of the dialog box by up to 0.5 pixels.

To see the correct view of the dialog box, use the shortcut Esc d x, Esc d w, or Esc d m. This redisplay the DRE file with the correct measurements.

Adding dialog items

Because dialog items are special Frame graphic objects, you cannot create them by using the Tools palette. To create a dialog item, you need to select an existing item in a DRE file and copy and paste the item.

When you add an item to a dialog box, the FrameMaker product assigns the next highest item number to the item.

Some dialog items, such as buttons, checkboxes, and radio buttons, already contain labels. You can add a label to dialog items that don't have labels, such as text boxes, pop-up menus, and scroll lists. Figure 10-6 shows that the label for a text box is a separate dialog item.

Some items, such as checkboxes, already have labels.



Since text boxes do not have labels, you need to add a separate label.



Figure 10-6 *Dialog item with a separate label*

A label is considered to be a separate dialog item if it is not part of a button, checkbox, or radio button. To add a label, select a separate label, then copy and paste it.

Deleting items from a dialog box

To delete an item from a dialog box, select the item and press the Delete key.

Note that by deleting an item, you break the sequence of item numbers. For example, if you delete item 3, the sequence of item numbers skips from 2 to 4. You need to renumber the items so that the sequence is unbroken.

To renumber the items in a dialog box, follow these steps:

- 1 *Select the dialog box.*
- 2 *From the Graphics menu, choose Object Properties.*
The FrameMaker product automatically renumbers the items.
- 3 *Click OK to dismiss the Dialog Box Properties dialog box.*

Setting the properties of the dialog box

The graphic object that represents a dialog box has special properties that you must set when you create a new DRE file.

To set the properties for a DRE file, follow these steps:

- 1 *In the DRE file, select the rectangle representing the dialog box.*
- 2 *From the Graphics menu, choose Object Properties.*

The Dialog Box Properties dialog box appears, displaying the properties of the dialog box. You can specify the following properties for a dialog box:

- The order of the items in the dialog box
- The title of the dialog box
- The item initially highlighted or selected by the cursor (called the *first focus* of the dialog box)
- The button activated by pressing the Return key (called the *default button*)
- The items that act as the OK, Cancel, and Help buttons
- The size and position of the dialog box (note that modal dialog boxes are always positioned in the center of the screen)

The Item Order list in the Dialog Box Properties dialog box displays a list of all the dialog items in the DRE file. Each dialog item is associated with an item number.

Note the item number of each dialog item. You use these item numbers to identify dialog items in your client code.

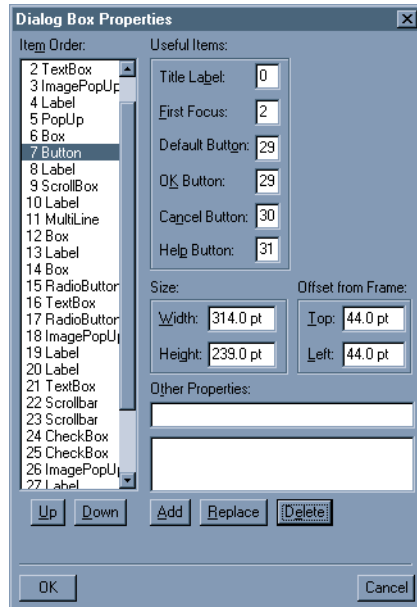


Figure 10-7 Properties of a dialog box

Specifying the focus order

When the user selects an item in a dialog box (except a label or a box), the item becomes the *focus* of the dialog box. For example, if the user clicks in a text box, the text box is the focus.

When the user presses the Tab key, the focus moves from one item to the next in a specific order (for example, if the cursor is in a text box and the user presses the Tab key, the focus might move to a button or pop-up menu). This order is called the *focus order* of a dialog box.

The focus order of a dialog box is specified by the order of items in the dialog box. For example, suppose item 4 is a text box and item 5 is a radio button. If the text box has the focus, pressing the Tab key moves the focus to the radio button.

The focus order defines the order in which items are selected when the user presses the Tab key. You can change the focus order by changing the order of items in the dialog box.

To change the focus order, follow these steps:

- 1 Select an item in the scroll list.
- 2 Click Up or Down to move the selected item up or down in the list.

To move the selected item up or down by 5 items at a time, press the Shift key while clicking Up or Down.

When you select an item in the Item Order scroll list, the corresponding graphic object in the DRE file also appears selected.

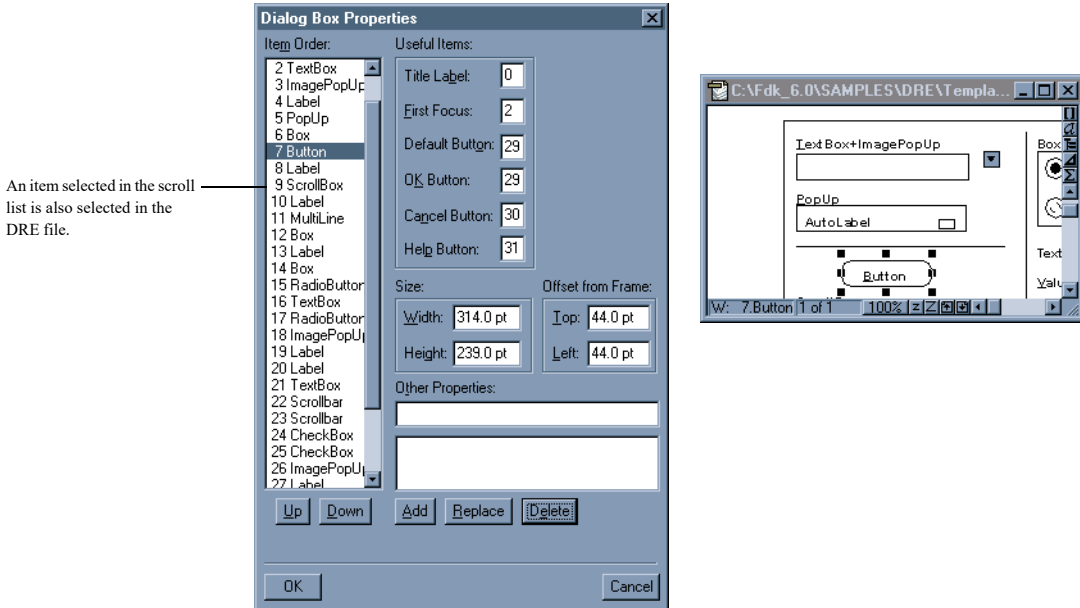


Figure 10-8 Item selected in the Item Order scroll list and the DRE file

If you assign a keyboard shortcut to an item, make sure that the label containing the shortcut and the item are in sequential order. Move the label item so that it precedes the item in the Item Order scroll list.

If you do not assign keyboard shortcuts, the label and the item do not need to be in sequential order.

If you use a box item to group other items, make sure the items it contains appear immediately after it in the Item Order scroll list. Note that this order is important only if the box contains other items; if the box does not contain any items or if the box is used as a separator (see “Boxes” on page 425), the item order is not important.

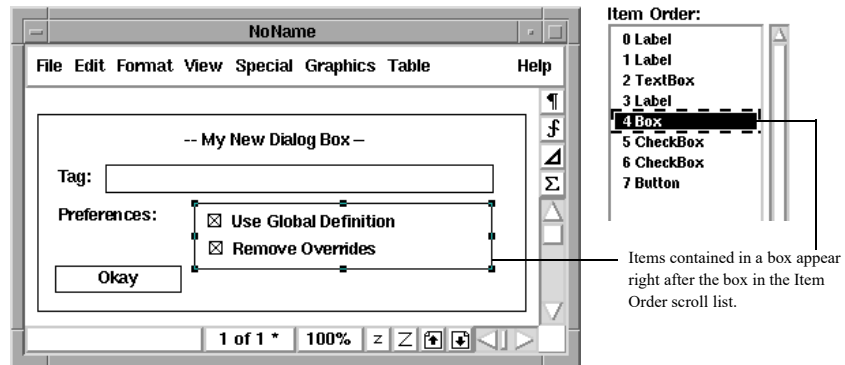


Figure 10-9 Order of items in the Item Order scroll list

Identifying basic items

Each dialog box requires some basic items. You need to specify these items as part of the properties of the dialog box. These items include:

- Title label
The *title label* is a label that appears in the title bar of the dialog box.
- First focus
The *first focus* of a dialog box is the item selected when the user first opens the dialog box.
- Default button
The *default button* is a button the user can activate by pressing the Return key.
- OK button
The OK button is the button that performs an action and dismisses the dialog box.
- Cancel button
If you are creating a modal dialog box, identify a Cancel button. If the user closes the dialog box by pressing Control-w, the dialog box activates the Cancel button.

Useful Items:

Title Label:	<input type="text" value="0"/>
First Focus:	<input type="text" value="2"/>
Default Button:	<input type="text" value="31"/>
OK Button:	<input type="text" value="31"/>
Cancel Button:	<input type="text" value="32"/>

Figure 10-10 Specifying the basic items in a dialog box

To specify any of these items, type the item number in the appropriate text box. For example, to specify the Cancel button as item 4, type 4 in the Cancel Button text box.

If a dialog box doesn't use one of these items, type -1 in the text box for the item. For example, to create a dialog box with no Help button, type -1 in the Help Button text box.

Specifying the size and position of a dialog box

You can change the properties of a dialog box that specify its initial size and position. You can also adjust the size of the dialog box manually by selecting its rectangle and dragging the handles.

Size:	Offset From Frame:
Width: <input type="text" value="404.0 pt"/>	Top: <input type="text" value="10.0 pt"/>
Height: <input type="text" value="227.0 pt"/>	Left: <input type="text" value="10.0 pt"/>

Figure 10-11 Specifying the size and position of a dialog box

The size and position values do not affect the initial position of modal dialog boxes. Modal dialog boxes always appear in the center of the screen.

Setting the properties of a dialog item

Like standard Frame graphic objects, dialog items have properties. These properties define the size and position of the item and relationships with other items.

You can set the following properties for a dialog item:

- Basic properties
- Properties specific to different types of dialog items
- Keyboard shortcuts

- Relationships with other dialog items

The rest of this section describes the different types of dialog items and explains how to set the properties for these items.

Setting basic dialog item properties

To set or modify an item's basic properties, follow these steps:

- 1 *Select the item.*
- 2 *From the Graphics menu, choose Object Properties.*
- 3 *In the Dialog Item Properties dialog box, specify the properties of the item.*

Figure 10-12 shows an example of the Dialog Item Properties dialog box.

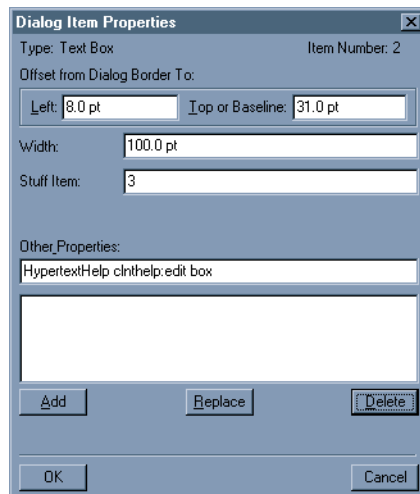


Figure 10-12 *Dialog Item Properties dialog box for a text box*

As is the case with standard Frame graphic objects, different dialog items have different properties. Each type of dialog item has a different Dialog Item Properties dialog box.

For most types of dialog items, you can set the following properties in the Dialog Item Properties dialog box:

- Position relative to the dialog box

In the Left text box, type the offset of the item from the left side of the dialog box.

In the Top or Baseline text box, type the offset of the item from the top of the dialog box.

- Width
In the Width text box, type the width of the item.
- Height
In the Height text box, type the height of the item.

Note that height is set to a fixed value for some items, such as text boxes, labels, and buttons. You cannot specify the height of any of these items. For these items, height is determined by the height of the text that appears in the item.

Similarly, you cannot specify the width of certain items, such as radio buttons, check boxes, and labels. For these items, width is determined by the length of the text that appears in the item.

For most types of items, the Dialog Item Properties dialog box also contains the Other Properties text box and the Stuff Item text box. The Stuff Item text box displays the number of a dialog item that is related to the current item. For instructions on relating dialog items, see “Relating items in a dialog box” on page 430.

Some item types have additional properties. Instructions on setting these properties are covered in the next section.

Working with specific types of items

The following sections describe each type of item and item properties not covered in the previous section. You set these properties in one of the following ways:

- Enter values in the Dialog Item Properties dialog box (for example, to specify the initial state of a checkbox)
- Select and edit the item in the DRE file (for example, to specify the text in a label, select the text and type over it)
- Write client code (for example, to specify the menu choices in a pop-up menu)

Boxes

A box appears as a rectangle drawn with a single black line. You can use a box to organize the items in a dialog box. The items contained within a box must have numbers that follow the item number of the box. For details, see “Specifying the focus order” on page 420.

You can also use a box as a line separator by setting its width or height to 0.

.....
IMPORTANT: *Do not put a box within another box, except in cases where the inside box is a separator (a box with a height or width of 0).*
.....

For information on setting the basic properties for this dialog item, see the section “Setting basic dialog item properties” on page 424.

Buttons

A button allows the user to execute a command or invoke an action from a dialog box.

The button item includes a label. To change the text in the label, select the text on the button face and type over it.

For information on setting the basic properties for this dialog item, see the section “Setting basic dialog item properties” on page 424.

Checkboxes

A checkbox allows the user to choose from two or three options. A checkbox can have two states (on or off) or three states (on, as is, or off). For example, checkboxes in the FrameMaker product Find/Change dialog box have two states; the checkboxes in the Character Designer have three states.

Three-state checkboxes are also called *triboxes*. Triboxes and two-state checkboxes are handled as different objects internally by FrameMaker products. For more information, see “How the API represents dialog boxes” on page 441.

In addition to the basic properties of the dialog item (see “Setting basic dialog item properties” on page 424), checkboxes have the following properties:

- Initial state of the checkbox
In the Initial State text box, type 0 if the checkbox is initially off or 1 if it is initially on.
- Number of states for the checkbox
In the States text box, type 2 if the checkbox has two states or 3 if the checkbox has three states.

The checkbox item includes a label. To change the text in the label, select the text next to the checkbox and type over it.

Radio buttons

A radio button allows the user to select one choice out of several choices. Each radio button belongs to a set. Only one radio button in a set can be selected at a time.

In addition to the basic properties of the dialog item (see “Setting basic dialog item properties” on page 424), radio buttons have the following properties:

- Initial state of the radio button (on or off)
In the Initial State text box, type 0 if the radio button is initially off or 1 if it is initially on.

- Number of the group to which the radio button belongs
In the Group text box, type the number that identifies the radio button's group. When you assign group numbers, start with 1.

The radio button item includes a label. To change the text in the label, select the text next to the radio button and type over it. If the text is longer than one line, add another label for each additional line of text.

You can set the label of the radio button in your client code. For details, see "Labels" on page 428.

Text boxes

A text box allows the user to enter text. The text box item does not include a label. For instructions on adding a label for a text box, see the section "Adding dialog items" on page 417.

For information on setting the basic properties for this dialog item, see the section "Setting basic dialog item properties" on page 424.

Multiline text boxes

A multiline text box is a text box that displays more than one line of text. The multiline text box item does not include a label. For instructions on adding a label for a multiline text box, see the section "Adding dialog items" on page 417.

For information on setting the basic properties for this dialog item, see the section "Setting basic dialog item properties" on page 424.

Pop-up menus

A pop-up menu allows the user to select a setting from a list of settings. You can't specify the list of settings in the DRE file. Instead, you must include code in your client to provide it. For more information, see "Initializing items in a dialog box" on page 449.

The pop-up menu dialog item does not include a label. For instructions on adding a label for a pop-up menu, see the section "Adding dialog items" on page 417.

For information on setting the basic properties for this dialog item, see the section "Setting basic dialog item properties" on page 424.

Image pop-up menus

An image pop-up menu is a pop-up menu that appears as a bitmap image of an arrow pointing downward. For an example of this bitmap image, look at the left side of the Paragraph Designer window.

In the Dialog Item Properties dialog box for this item, the name of the bitmap used is specified in the File Name text box. The name can be either `arrowdown` or `tallarrowdn`:

- These two bitmaps are identical. Do not change these names.

You can't specify the list of settings for an image pop-up menu in the DRE file. Instead, you must include code in your client to provide it. For more information, see "Initializing items in a dialog box" on page 449.

The image pop-up menu item does not include a label. For instructions on adding a label for an image pop-up menu, see the section "Adding dialog items" on page 417.

For information on setting the basic properties for this dialog item, see the section "Setting basic dialog item properties" on page 424.

Labels

A label is a single line of text that you can use to identify other items in a dialog box. Some types of dialog items, such as radio buttons, checkboxes, and buttons, already include labels. For these items, the label is part of the dialog item. Other items, such as text boxes and pop-up menus, do not include labels. For these items, the label is a separate dialog item.

To change the text in a label, select the label's text in the DRE file and type over it. You can also include code in your client to change a label dynamically. To do this, set the label property on the API object representing the label. For details on the properties of the API object, see "Dialog boxes" in the FDK Programmer's Reference. For instructions on getting and setting properties, see Chapter 5, "Getting and Setting Properties."

If you set the text of a label in your client code, you must make sure that the DRE file defines an adequate space for the text. Otherwise, the text set by your client may appear truncated. For example, suppose you add a label to a DRE file and specify *String* as the text in the label. In your client code, if you set this label to *String of text*, the actual label in the dialog box only displays the word *String* and truncates the rest of the label (*of text*).

To prevent this, select the label and type the longest string of text set by your client code. If you do not want this string of text to appear when the dialog box is displayed, you can initialize the value of the label in your client code so that the long string is not displayed by default. For examples of initializing the values of dialog items, see "Initializing items in a dialog box" on page 449.

For information on setting the basic properties for this dialog item, see the section "Setting basic dialog item properties" on page 424.

Scroll bars

A scroll bar allows the user to choose a value within a specified range. Each end of the scroll bar represents one end of the range. You specify a scroll bar's range in your client code. To do this, set the minimum and maximum value properties on the API object representing the scroll bar. For details on the properties of the API object, see "Dialog boxes" in the FDK Programmer's Reference. For instructions on getting and setting properties, see Chapter 5, "Getting and Setting Properties."

To display the currently selected value of a scroll bar in the dialog box, add a text box or a label. The dialog box does not automatically stuff the scroll bar value into the text box or label. To do this, you must add code to your client that gets the scroll bar value and programmatically stuffs it into the text box or label.

If you resize a scroll bar by dragging on its object handles, it may appear distorted. To eliminate the distortion, use the keyboard shortcut for viewing the dialog box on the current platform. For instructions on using this shortcut, see "Set the properties of the dialog box" on page 437.

To change the length of a scroll bar, type over the value in the Width or the Height text box:

- If the scroll bar is horizontal, type the length of the scroll bar in the Width text box.
- If the scroll bar is vertical, type the length of the scroll bar in the Height text box.

Note that you cannot change the width of a vertical scroll bar. Similarly, you cannot change the height of a horizontal scroll bar.

To change the orientation of a scroll bar from horizontal to vertical, drag the object handles of the scroll bar so that the height of the scroll bar is greater than the width. The scroll bar appears distorted until you use the keyboard shortcut for viewing the dialog box on the current platform. You can use a similar process to change the orientation from vertical to horizontal.

For information on setting the basic properties for this dialog item, see the section "Setting basic dialog item properties" on page 424.

Scroll lists

A scroll list is a list of items from which the user can select an item. You can't specify the list of items in the DRE file. Instead, you must include code in your client to provide it. For more information, see "Initializing items in a dialog box" on page 449.

The scroll list item does not include a label. For instructions on adding a label for a scroll list, see the section "Adding dialog items" on page 417.

For information on setting the basic properties for this dialog item, see the section "Setting basic dialog item properties" on page 424.

Specifying keyboard shortcuts for Windows versions

In Windows, the user can activate or select certain dialog items by pressing a keyboard shortcut. The shortcut is identified by the first underlined letter in the item's label. When the user presses the Alt key and this letter, the dialog item is activated.

Figure 10-13 illustrates how a shortcut is identified in the label of a text box.

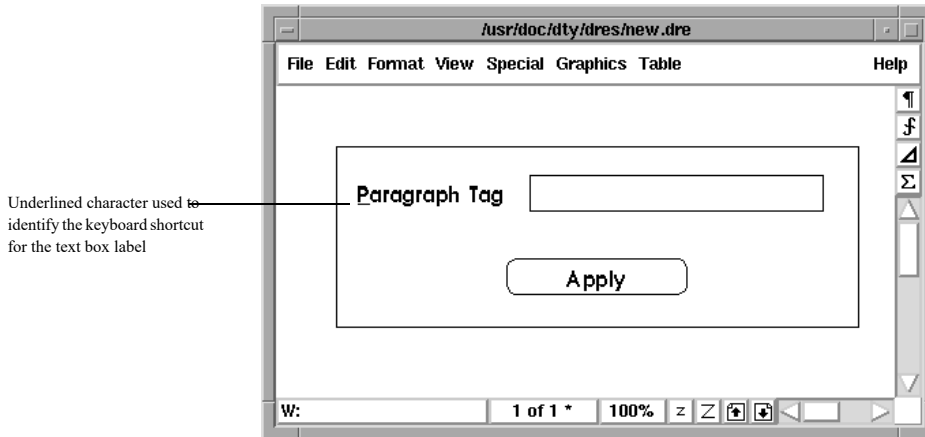


Figure 10-13 Keyboard shortcut for an item in Windows

To specify a keyboard shortcut for the Windows version of a dialog box, follow these steps:

- 1 In the DRE file, select the letter of the label to be used for the keyboard shortcut.
- 2 From the Format menu, choose *Style>Underline*.

If the item has a separate label item, make sure that its item number follows the label's item number. For example if the item is a text box, make sure its item number follows the item number of its label. For more information, see “Specifying the focus order” on page 420.

Relating items in a dialog box

You can relate certain types of dialog items to other items. If two items are related, manipulating one changes the other.

For example, in the FrameMaker product Paragraph Designer, the Paragraph Tag pop-up menu consists of two dialog items: a text box and an image pop-up menu. The image pop-up menu is the bitmap image of an arrow pointing downward. If the user chooses a setting from the image pop-up menu, the setting is displayed, or *stuffed*, in the text box. The image pop-up menu's *stuff item* is the text box.

Figure 10-14 shows a text box and an image pop-up menu. The text box is the stuff item of the image pop-up menu.

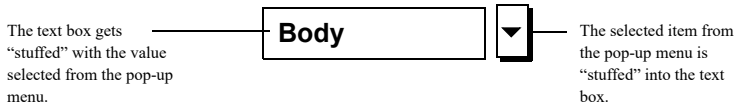


Figure 10-14 *Related dialog items*

A text box can also be related to a radio button. If a radio button's stuff item is a text box, turning on the radio button moves focus to the text box. If the stuff item for a text box is a radio button, typing in the text box automatically turns on the radio button.

To specify the stuff item for a dialog item, follow these steps:

- 1 *In the DRE file, select the item.*
For example, if you want a text box to be the stuff item for an image pop-up menu, select the image pop-up menu first. The order in which you select the items is important.
- 2 *Select the item which will be the stuff item of the first item.*
- 3 *To identify the second item you selected as the stuff item for the first item you selected, press Esc d f.*

The FrameMaker product automatically deselects the second item. The first item remains selected, and the status bar displays information about this item. The item number of the second item is displayed in parentheses, indicating that the second item is the stuff item for the first item.

For example, suppose you select an image pop-up menu (item 18) first, then you select a text box (item 17). Then you use the keyboard shortcut Esc d c. The text box becomes the stuff item of the image pop-up menu. The FrameMaker product deselects the text box. The image pop-up menu remains selected, and the status bar displays the following information:

```
X:18 . ImagePopup (17)
```

The number in parentheses indicates that item 17 is the stuff item of item 18.

You can also specify the stuff item by choosing Object Properties from the Graphics menu and typing the number of the related item in the Stuff Item text box.

Saving a DRE file

.....
IMPORTANT: To save the DRE file, choose the Save command from the File menu. To save it under a different name, choose the Save As command from the File menu.
.....

The Save As dialog box appears with the option to save the files as a Frame dialog resource. This option is only available when you open a DRE file in a FrameMaker product. If you open a Frame binary file, this option does not appear.

Saving a DRE file creates additional files—Windows dialog resource files (.dlg) and extra dialog information files (.xdi files). You compile these files with FDK client.

When a FrameMaker product creates these additional files, it names these files after the DRE file. For example, saving the file named `mydlg.dre` creates additional files named `mydlg.dlg` and `mydlg.xdi`.

The dialog resource is named after the base name of the DRE file (the filename without the .dre extension).

When you open the dialog resource using the `F_ApiOpenResource()` function, use the name for the resource as an argument to the function. The name of the resource is usually the base name of the DRE file.

For details on opening dialog resources and displaying dialog boxes, see “Opening dialog resources” on page 448.

Modeless Dialog Boxes

FrameMaker FDK provides various ways to create and manage modeless dialog boxes.

Modeless dialog boxes in Workspaces

Because FrameMaker provides support for workspaces, the client’s modeless dialogs can become a part of a workspace. To make this work, the client has to handle the notification `FA_Note_Dialog_Create`, which is sent to the client when the workspace has to launch the modeless dialog for a particular client.

When the user closes a client’s modeless dialog, the dialog event `FV_DlgClose` is issued. Note that this dialog can also be closed due to workspace-related operations, such as switching workspaces. In such cases, instead of the dialog event `FV_DlgClose`, the notification `FA_Note_QuitModelessDialog` is sent to the client. Therefore, the client must handle both of these events appropriately to achieve the desired result.

Within a workspace, if the dialog gets visible from a minimized/iconic state, then a dialog event `FV_DlgNeedsUpdate` is issued. This event indicates that the client’s modeless dialog has become visible and should be updated, so that it does not display stale information.

Capturing behavior of a modeless dialog box and controlling its position

A client can capture behavior of modeless dialog and control its position. For modeless dialogs, client can handle the dialog's show-hide behavior using the events FV_DlgHide and FV_DlgShow. It can also handle its close behavior by capturing event FV_DlgClose and control its hide behavior on close. Position of the modeless dialog can be controlled by DLG(.dlg file) and its docking can be controlled too. The following table details the FDK support for modeless dialog:

Functionality	FDK support
Open a modeless dialog as docked or undocked	<pre>F_ApiSetInt (session_id, dialog_id, FP_DockDialog, Dock_value);</pre> <p>Dock_value can be one of the following:</p> <pre>FV_DIALOG_DOCK_NONE FV_DIALOG_DOCK_BOTTOM FV_DIALOG_DOCK_RIGHT FV_DIALOG_DOCK_LEFT</pre>
Determine if a dialog is docked or undocked	<pre>F_ApiGetInt (session_id, dialog_id, FP_IsDialogDocked);</pre>
Determine if a dialog is displayed	<pre>F_ApiGetInt (session_id, dialog_id, FP_IsDialogVisible);</pre>
Control dialog on hide/close behavior	<p>Client can capture the FV_DlgClose event. To hide a dialog on close, use the following returnvalue:</p> <pre>F_ApiReturnValue (FR_HideDialogOnClose);</pre>
Control the position of the dialog	<p>The position of the dialog can be controlled using .dlg file.</p>
Control the position of the dialog	<p>Events FV_DlgHide & FV_DlgShow</p>

Testing a dialog box

You can test a dialog box while you are modifying it to verify its appearance and its item focus order. You can test it as a modal dialog box or as a modeless dialog box.

- A modal dialog box prevents the user from performing any other action in a FrameMaker product while the dialog box is visible. For example, the Print dialog box is a modal dialog box. Until the user clicks Print or Cancel to close the dialog box, the user can't perform any other action.
- A modeless dialog box allows the user to perform other actions in a FrameMaker product while the dialog box is displayed. For example, the Marker dialog box is a modeless dialog box. While the Marker dialog box is displayed, the user can do other work in a Frame document.

A DRE file does not specify whether a dialog box is modal or modeless. Your client code determines the dialog box type when it displays it. For more information, see “Displaying a dialog box” on page 450.

To test a dialog box, use one of the following keyboard shortcuts:

- To test the dialog box as modal, press Esc d t.
In this mode, clicking any button dismisses the dialog box.
- To test the dialog box as modeless, press Esc d T.
In this mode, clicking any button does not dismiss the dialog box. To close the dialog box, use the native window manager functionality.

Figure 10-15 shows a DRE file and the dialog box displayed when it is tested.

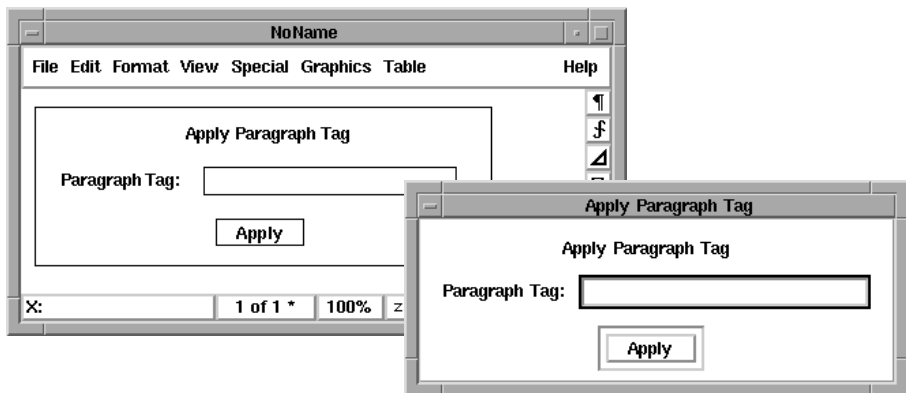


Figure 10-15 Dialog box being tested

A simple example

This section provides an example of how to create a simple dialog box named `pgftag`. For instructions on writing the client code to implement this dialog box, see “A simple example” on page 445.

The dialog box is shown in Figure 10-16.



Figure 10-16 The `pgftag` dialog box

To create the sample dialog box, follow these general steps:

1 *Create a new DRE file.*

For more information, see “Creating a DRE file” on page 412.

2 *Design the layout of the dialog box.*

3 *Set the properties of the dialog box.*

4 *Save and test the DRE file.*

Note that this example does not involve setting properties for specific dialog items, since the example is relatively simple.

The steps for creating the sample dialog box are described in the following sections.

Designing the layout of the dialog box

To design the layout of the sample dialog box, follow these steps:

- 1 Delete all extraneous items from the DRE file, except two labels, a text box, and a button.
- 2 Drag the object handles of the dialog box rectangle to resize it.
- 3 Select the text in one of the label items and type `Apply Paragraph Tag`.
- 4 Select the text in the other label item and type `Paragraph Tag`.
- 5 Select the text in the button's label and type `Apply`.
- 6 Drag the object handles of the text box and the button to resize them.
- 7 Position the items within the dialog box.

Figure 10-17 shows the DRE file with the layout of the dialog box completed.

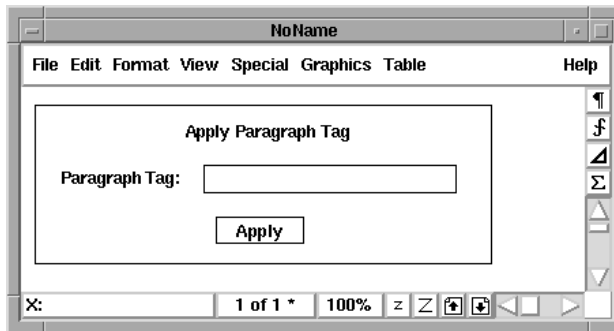


Figure 10-17 Completed layout of the dialog items

- 8 If you intend to create the dialog box for more than one platform, change the platform view to check the layout on the other platforms.
You may need to make minor adjustments to the layout. If you do, you need to apply the changes to the other platform views. Press `Esc d r` to reset the other platform views.

Set the properties of the dialog box

To set the properties of the sample dialog box, follow these steps:

- 1 Select the dialog box in the new DRE file.
- 2 From the *Graphics* menu, choose *Object Properties*.
The Item Order scroll box lists the three items remaining in this DRE file.

- 3 *Using the Up and Down buttons, change the order of items in the dialog box so that the “Apply Paragraph Tag” label is item 0, the button is item 1, the text box is item 2, and the “Paragraph Tag” label is item 3.*

These numbers are used to identify the dialog items in the code. For details on the code used to run this example, see “A simple example” on page 445.

Note that the label for the text box follows the text box in consecutive order. Since no keyboard shortcuts are defined in this example, the label of the text box does not need to precede the text box. The label and the text box can be in any order and do not need to be consecutive in the Item Order scroll box.

- 4 *In the Useful Items group box, verify that the following items are specified:*

- First Focus is set to 2.
- Title Label is set to 0.
- Default Button is set to 1.
- OK Button is set to 1.

Since the dialog box has only one button, you do not need to specify the Cancel button.

- 5 *Click OK to dismiss the Dialog Box Properties dialog box.*

Saving and testing the DRE file

When you finish designing the sample dialog box, you need to test and save it:

- 1 To save the file as a DRE file, choose *Save As* from the *File* menu.
- 2 Type the name:
`pgftag.dre`

Two additional files are created: `pgftag.dlg` and `pgftag.xdi`.

You can use these files with the client code for this dialog box (see “A simple example” on page 445) to build an FDK client.

- 3 Press `Esc d t` to test the dialog box.

General tips for dialog editing

When editing dialog boxes, keep the following in mind:

- A DRE file can contain only one dialog box. If the DRE file already contains a dialog box, do not paste another dialog box into the DRE file.
- In the Dialog Item Properties dialog box, FrameMaker products display size and position information in units of points. You cannot switch the units used for display, even by specifying a change in the View Options dialog box. You can, however, use any units to enter data.

Summary of keyboard shortcuts

The following table lists the keyboard shortcuts for editing dialog boxes.

To do this:	Use this shortcut:
Relate the selected items to each other	<code>Esc d f</code>
Select the first item in the draw order (usually, this is the dialog box in the DRE file)	<code>Esc o F</code>
Select the next item in the draw order	<code>Esc o n</code>
Display the properties of the selected dialog box or item	<code>Esc g o</code>
Test the dialog box as modal	<code>Esc d t</code>
Test the dialog box as modeless	<code>Esc d T</code>

To do this:	Use this shortcut:
View the dialog box	Esc d w
Clear the geometry information from local memory	Esc d r

Handling Custom Dialog Box Events

.....

.....

This chapter describes how to use custom dialog boxes in your client’s user interface. For instructions on creating custom dialog boxes, see Chapter 10, “Creating Custom Dialog Boxes for Your Client.”

If your client’s user interface requires only simple modal dialog boxes, you may not need to create or use custom dialog boxes. The API provides several simple, ready-made modal dialog boxes. For information on using these dialog boxes, see “Using API dialog boxes to prompt the user for input” on page 195.

How the API represents dialog boxes

The API uses an `FO_DialogResource` object to represent each dialog resource in a FrameMaker product session. It also uses an object to represent each item in a dialog resource. The following table lists the types of dialog items and the types of objects the API uses to represent them.

Dialog item	API object type
Box	<code>FO_DlgBox</code>
Button	<code>FO_DlgButton</code>
Checkbox	<code>FO_DlgCheckBox</code>
Image pop-up menu	<code>FO_DlgImage</code>
Label	<code>FO_DlgLabel</code>
Pop-up menu	<code>FO_DlgPopUp</code>
Radio button	<code>FO_DlgRadioButton</code>
Scroll list	<code>FO_DlgScrollBar</code>
Text box or multiline text box	<code>FO_DlgEditBox</code>
Three-state checkbox (tribox)	<code>FO_DlgTriBox</code>

The following table lists some dialog item properties.

Property	Type	Meaning
FP_Label	StringT	The label that appears adjacent to the item.
FP_Labels	F_StringST	If the item is a scroll list, pop-up menu, or image pop-up menu, the list of strings it contains.
FP_Sensitivity	IntT	Specifies whether the item is enabled. If <code>FP_Sensitivity</code> is <code>False</code> , the item is disabled and appears dimmed.
FP_State	IntT	The state of the item. If the item is a pop-up menu, image pop-up menu, or scroll list, <code>FP_State</code> specifies the index (in the list specified by the <code>FP_Labels</code> property) of the chosen string. If no string is chosen, <code>FP_State</code> is <code>-1</code> . If the item is a button, a checkbox, or a radio button, <code>FP_State</code> specifies <code>FV_DlgOptNotActive</code> when the item is off and <code>FV_DlgOptActive</code> when the item is on. If the item is a tribox, <code>FP_State</code> can also specify <code>FV_DlgOptDontCare</code> when the item is set to <code>As Is</code> .

Not all types of dialog items have all of these properties. For a complete list of properties for each type of dialog item, see “Dialog boxes” in the FDK Programmer’s Reference.

Dialog resource and dialog item IDs

When you open a dialog resource, the API returns its ID. The API also assigns a unique ID to each item in a dialog box. To get a dialog item’s ID, call

```
F_ApiDialogItemId().
```

The syntax for `F_ApiDialogItemId()` is:

```
F_ObjHandleT F_ApiDialogItemId(F_ObjHandleT dialogId,
    IntT itemNum);
```

This argument	Means
<code>dialogId</code>	The ID of the dialog box containing the item
<code>itemNum</code>	The item number of the item

`F_ApiDialogItemId()` returns the dialog item's ID or 0 if the item doesn't exist.

A dialog item's item number appears in the Dialog Object Properties window for the item when you create the dialog box. It also appears in the Frame dialog resource file following the description of the item. For example, the following portion of a Frame dialog resource file describes a text box item with the item number 2:

```
<EditBox
  <BaseLine 23 74 80>
  <Label ImATextBox>
  <StuffObject -1>
  <HypertextHelp dbre.hlp:edit box>
> # 2
```

If the name of the dialog resource containing this text box is `mydlg.x`, you can use the following code to get its ID:

```
. . .
#define EDITBOX_ITEM_NUM 2
F_ObjHandleT dlgId, editboxId;
dlgId = F_ApiOpenResource(FO_DialogResource, "mydlg");
editboxId = F_ApiDialogItemId(dlgId, EDITBOX_ITEM_NUM);
. . .
```

Getting and setting dialog item properties

You can get and set the dialog item properties with `F_ApiGetPropertyType()` and `F_ApiSetPropertyType()` functions. When you call one of these functions, set its first parameter to the

dialog resource ID and its second parameter to the ID of the item for which you want to get or set a property. For more information on using `F_ApiGetPropertyType()` and `F_ApiSetPropertyType()` functions, see Chapter 5, "Getting and Setting Properties."

For example, if you create a dialog resource named `mydlg`, which contains a checkbox with the item number 3, the following code opens the resource and turns the checkbox on:

```
. . .
#define CHECKBOX_ITEM_NUM 3
F_ObjHandleT dlgId;
dlgId = F_ApiOpenResource(FO_DialogResource, "mydlg");
F_ApiSetInt(dlgId, F_ApiDialogItemId(dlgId, CHECKBOX_ITEM_NUM),
            FP_State, FV_DlgOptActive);
. . .
```

Manipulating related items

When you change a dialog box programmatically, the dialog box behaves as if you are changing it interactively. For example, if you programmatically turn on one radio button in a set, the radio button in the set that was previously turned on automatically turns off. If a text box is the stuff item for a pop-up menu, when you choose a setting in the pop-up menu, the item is automatically stuffed in the text box.

Overview of using a custom dialog box in your client

To use a custom dialog box in your client, follow these general steps:

1 *Call `F_ApiOpenResource()` to open the dialog resource.*

Set the first parameter of `F_ApiOpenResource()` to `FO_DialogResource` and the second parameter to the name of the dialog box. `F_ApiOpenResource()` returns the dialog resource's ID.

2 *Add code to initialize items in the dialog box.*

A dialog resource does not provide default settings or values for many types of dialog items. For example, scroll lists, pop-up menus, and image pop-up menus are empty when you first open the dialog resource. If you want any items to have default settings, call `F_ApiSetPropertyType()` functions to provide them after you call `F_ApiOpenResource()` but before you display the dialog box.

3 *Add code to display the dialog box.*

To display a modal dialog box, call `F_ApiModalDialog()`. To display a modeless dialog box, call `F_ApiModelessDialog()`.

4 *Add code to update the dialog box.*

If the dialog box is modeless, you may want to update it when the user changes things in the FrameMaker product session. For example, if the dialog box displays a scroll list of all the open documents in a FrameMaker product session, you may want to update the list whenever the user opens or closes a document.

To update a dialog box, turn on notifications, such as `FA_Note_BackToUser` and `FA_Note_PostFunction`. Then add code to your client's `F_ApiNotify()` callback to set item properties when it receives these notifications.

5 *Add code to respond to user actions in the dialog box.*

How you handle user actions in a dialog box depends on how you display the dialog box. If you display it as a modeless dialog box, you should add a callback function named `F_ApiDialogEvent()` to your client. The FrameMaker product attempts to call this function whenever the user manipulates the dialog box. Your client's `F_ApiDialogEvent()` function can call API functions to get or set the properties of items in the dialog box. It can also get and set the properties of other objects in the session and call functions to execute operations, such as opening and closing documents.

If you display a dialog box as a modal dialog box, you can instruct the API to call `F_ApiDialogEvent()` for each dialog event just as it would for a modeless dialog box, or you can wait until the dialog box is closed and then check the properties of the dialog's items to determine what the user changed.

6 *Add code to respond to the user closing the dialog box.*

The user can close a dialog box by pressing Control-w . When the user closes the dialog box, the FrameMaker product calls your client's `F_ApiDialogEvent()` function. Your client may need to conduct some special processing in response to this call. For example, it may need to turn off notifications that it uses for updating the dialog box.

The following sections discuss these steps in greater detail.

A simple example

The following client implements a custom dialog box named `pgftag` as a modeless dialog box. For instructions on creating this dialog box, see "A simple example" on page 435.

The `pgftag` dialog box contains a text box that displays the paragraph tag of the paragraph containing the insertion point. The user can change the paragraph tag by typing a different tag in the text box and clicking the Apply button.

Following the code is a line-by-line description of how it works.

```

1  #include "fapi.h"
2  #define APPLY_BUTTON 1
3  #define TAG_FIELD 2
4  #define DLG_NUM 1
5  F_ObjHandleT dlgId = 0;
6
7  VoidT F_ApiInitialize(init)
8      IntT init;
9  {
10     dlgId = F_ApiOpenResource(FO_DialogResource, "pgftag");
11     F_ApiModelessDialog(DLG_NUM, dlgId);
12     F_ApiNotification(FA_Note_BackToUser, True);
13 }
14
15 VoidT F_ApiNotify(notification, docId, filename, iparm)
16     IntT notification;
17     F_ObjHandleT docId;
18     StringT filename;
19     IntT iparm;
20 {
21     F_TextRangeT tr;
22     StringT tag;
23
24     /* Get tag of first paragraph in selection. */
25     docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
26     tr = F_ApiGetTextRange(FV_SessionId, docId, FP_TextSelection);
27     if(!tr.beg.objId) return; /* No text selected in document. */
28     tag = F_ApiGetString(docId, tr.beg.objId, FP_Name);
29
30     F_ApiSetString(dlgId, F_ApiDialogItemId(dlgId, TAG_FIELD),
31                   FP_Text, tag); /* Stuff tag in text box. */
32     F_Free(tag);
33 }
34
35 VoidT F_ApiDialogEvent(dlgNum, itemNum, modifiers)
36     IntT dlgNum;
37     IntT itemNum;
38     IntT modifiers;
39 {
40     F_TextRangeT tr;
41     F_ObjHandleT docId;
42     StringT tag;
43
44     if (itemNum == FV_DlgClose) /* User closed dialog box. */
45         F_ApiNotification(FA_Note_BackToUser, False);
46
47     if(itemNum != APPLY_BUTTON) return; /* Apply not pressed. */

```

```
48
49     tag = F_ApiGetString(dlgId, F_ApiDialogItemId(dlgId,
50                               TAG_FIELD), FP_Text);
51     if(!tag || F_StrLen(tag) < 1) return; /* Text box empty */
52
53     /* Get current selection and apply tag to first pgf in it. */
54     docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
55     tr = F_ApiGetTextRange(FV_SessionId, docId, FP_TextSelection);
56     if(!tr.beg.objId) return;
57     F_ApiSetString(docId, tr.beg.objId, FP_Name, tag);
58     F_Free(tag);
59 }
```

Line 5

This line declares the ID of the dialog resource. It is declared as a global variable because the FrameMaker product does not pass dialog IDs to the `F_ApiDialogEvent()` callback. You need the ID to get and set properties of the items in the dialog box.

Lines 10 to 11

These lines open and display the dialog box when the FrameMaker product starts. The call to `F_ApiOpenResource()` opens the dialog resource. The call to `F_ApiModelessDialog` displays the dialog box as a modeless dialog box. `DLG_NUM` is a unique number used to identify the dialog box. It is passed to the `F_ApiDialogEvent()` callback when the user does something in the dialog box. If your client uses multiple modeless dialog boxes, you can use the dialog number passed to `F_ApiDialogEvent()` to determine which dialog box an event has occurred in.

Line 12

This line turns on the `FA_Note_BackToUser` notification. The FrameMaker product sends this notification to the client each time it finishes processing some user input. For example, each time the user changes the insertion point or applies a paragraph format, the FrameMaker product sends this notification to the client.

Lines 26 to 28

These lines handle the `FA_Note_BackToUser` notification. When the client receives this notification, it is possible the user has changed the insertion point or applied a new tag to the paragraph containing the insertion point. To make sure the dialog box is displaying the correct tag, the client gets the tag of the paragraph containing the insertion point each time it receives the notification.

Line 30

This line uses `F_ApiSetString()` to stuff the paragraph tag into the text box. When you get or set dialog item properties, you must specify a dialog box ID and a dialog item ID. The call to `F_ApiDialogItemId()` gets the ID of the text box.

Lines 35 to 59

These lines define the `F_ApiDialogEvent()` callback. When a user changes an item in a client-defined dialog box, the FrameMaker product calls this function with `dlgNum` set to the dialog box's number, `itemNum` set to the number of the item, and `modifiers` set to bit flags specifying the modifier keys the user was holding down.

Lines 44 to 45

These lines check `itemNum` to determine whether the user closed the dialog box. When the user closes the dialog box, the FrameMaker product sets `itemNum` to `FV_DlgClose`. If the dialog box was closed, the client no longer needs the `FA_Note_BackToUser` notification and can turn it off.

Line 47

This line checks `itemNum` to determine whether the user clicked Apply. If the user did anything but click Apply, the client ignores it and returns.

Lines 49 to 58

These lines get the string in the text box and change the tag (`FP_Name` property) of the current paragraph to it. Note that programmatically changing a paragraph tag does not change other properties of the paragraph.

Opening dialog resources

To open a dialog resource, call `F_ApiOpenResource()`.

The syntax for `F_ApiOpenResource()` is:

```
F_ObjHandleT F_ApiOpenResource(IntT resourceType,
    StringT resourceName);
```

This argument	Means
<code>resourceType</code>	The type of resource to open. To open a dialog resource, specify <code>FO_DialogResource</code> .
<code>resourceName</code>	The name of the resource to open. Specify the resource name.

`F_ApiOpenResource()` looks for the specified dialog resource first in the client resources, that is the DLL.

.....
IMPORTANT: *Your client's dialog resources must be correctly set up for the API to open them.*
.....

If `F_ApiOpenResource()` doesn't find the dialog resource in your client's resources, it looks in the FrameMaker product's resources. If it can't find the dialog resource in either the client or the FrameMaker product resources, it returns 0. If it finds and successfully opens the dialog resource, it returns the dialog resource's ID.

.....
IMPORTANT: *To manipulate a dialog box outside of the function in which you opened it, you must save its ID to a global variable.*
.....

For example, the following code opens a dialog box named `mydlg`:

```
. . .  
F_ObjHandleT dlgId; /* Global declaration */  
. . .  
dlgId = F_ApiOpenResource(FO_DialogResource, "mydlg");  
. . .
```

Initializing items in a dialog box

Before you display a dialog box, you can initialize the state or value of the items it contains. The dialog resource provides defaults for most items. However, it does not provide defaults for `FO_DlgPopup`, `FO_DlgScrollBar`, and `FO_DlgImage` items. Be sure to include code to initialize these items.

When you set the `FP_Labels` property of an `FO_DlgPopup` item, set the first string in the `F_StringsT` structure to the title of the pop-up menu.

When you first open a dialog box containing an `FO_DlgPopup` item, the default state of the item is 0, so the title appears as the current choice. Because the title is not a valid choice, you should initialize the `FP_State` property of `FO_DlgPopup` items to another number.

For example, the following code initializes a pop-up menu:

```
. . .
F_ObjHandleT dlgId, popupMenuId;
F_StringsT strings;
strings.val = (StringT *) F_Alloc(3*sizeof(StringT), NO_DSE);
strings.len = 3;

strings.val[0] = F_StrCopyString("Popup title");
strings.val[1] = F_StrCopyString("PopupItem1");
strings.val[2] = F_StrCopyString("PopupItem2");
F_ApiSetStrings(dlgId, popupMenuId, FP_Labels, &strings);

/* Make the first item the default. */
F_ApiSetInt(dlgId, popupMenuId, FP_State, 1);
. . .
```

When you initialize a dialog box, you may also want to disable some of its items, making them unavailable to the user. A disabled item appears dimmed and can't receive input focus. To disable an item, set its `FP_Sensitivity` property to `False`. For example, the following code disables a dialog item:

```
. . .
F_ObjHandleT dlgId, itemId;
F_ApiSetInt(dlgId, itemId, FP_Sensitivity, False);
. . .
```

Displaying a dialog box

After you have opened a dialog resource, you can display it as a modal or a modeless dialog box. To display it as a modal dialog box, call `F_ApiModalDialog()`. To display it as a modeless dialog box, call `F_ApiModelessDialog()`.

The syntax for `F_ApiModalDialog()` and `F_ApiModelessDialog()` is:

```
IntT F_ApiModalDialog(IntT dlgNum,
    F_ObjHandleT dlgId);

IntT F_ApiModelessDialog(IntT dlgNum,
    F_ObjHandleT dlgId);
```

This argument	Means
<code>dlgNum</code>	A unique number to identify the dialog box. The API passes this number to your client's <code>F_ApiDialogEvent()</code> callback when there is a user action in the dialog box. If you don't want the API to call your client's <code>F_ApiDialogEvent()</code> callback when there is a user action, set <code>dlgNum</code> to 0.
<code>dlgId</code>	The ID of the dialog resource to display.

`F_ApiModelessDialog()` returns immediately. If it can't display the dialog box, it returns an error code. Otherwise, it returns `FE_Success`.

If you call `F_ApiModalDialog()` with `dlgNum` set to 0, it does not return until the user closes the dialog box. If the user clicks Help in the dialog box, `F_ApiModalDialog()` returns a nonzero value; otherwise, it returns `FE_Success`.

If you set `dlgNum` to a nonzero value, when the user manipulates the dialog box, the API calls your client's `F_ApiDialogEvent()` callback, just as it does for a modeless dialog box.

Updating items in a dialog box

If you display a modeless dialog box, you may need to request and monitor notifications to update it when the user changes things in the FrameMaker product session. For example, if the dialog box displays information that is dependent on the insertion point, you should request and monitor the `FA_Note_BackToUser` or `FA_Note_PostFunction` notifications so that you can update the dialog box whenever the user changes the insertion point.

Avoid requesting more notifications than you need to update a dialog box. Also, be sure to turn off notifications after the dialog box is closed. If your client requests notifications that are issued very frequently, it can decrease FrameMaker product performance.

For example, if your dialog box includes a scroll list that displays a list of open documents in the session, you could update the list by requesting and monitoring the `FA_Note_BackToUser` notification. However, this would be inefficient because the FrameMaker product would issue notifications for many events that don't affect the

dialog box. It is much more efficient to request and monitor only the `FA_Note_PostOpenDoc` and `FA_Note_QuitDoc` notifications.

If you need to monitor notifications that are issued very frequently, such as `FA_Note_BackToUser` or `FA_Note_PostFunction`, avoid conducting extensive processing each time they are issued. If you don't, you may decrease FrameMaker product performance.

For example, suppose you request the `FA_Note_PostFunction` notification so that you can update a dialog box when the insertion point changes. The API issues the `FA_Note_PostFunction` notification for nearly every event in a FrameMaker product session. If you update the entire dialog box every time you receive the notification, it slows the FrameMaker product. It is more efficient to first determine whether the insertion point changed, and then to update the dialog box only if it changed.

Handling user actions in dialog boxes

How you handle user actions in a dialog box depends on whether the dialog is modal or modeless. The following sections discuss how to handle user actions in each type of dialog box.

Handling user actions in a modeless dialog box

After you display a modeless dialog box, the API attempts to call a function named `F_ApiDialogEvent()` from your client whenever the user does something, such as click the mouse or press a key, in the dialog box.

Your client should define `F_ApiDialogEvent()` as follows:

```
VoidT F_ApiDialogEvent(IntT dlgNum,
    IntT itemNum,
    IntT modifiers);
```

This argument	Means
<code>dlgNum</code>	The number of the dialog box in which the user action occurred (that is, the number you specified when you displayed the dialog box with <code>F_ApiModelessDialog()</code> or <code>F_ApiModalDialog()</code>).
<code>itemNum</code>	If the user manipulated a specific dialog item, <code>itemNum</code> is a nonnegative number specifying the dialog item. If the user didn't manipulate a specific dialog item, <code>itemNum</code> is a negative number constant specifying what the user did. For example, if the user closed the dialog box, <code>itemNum</code> is set to <code>FV_DlgClose</code> . For a list of the constants, see "Handling special events in a modeless dialog box" on page 460.
<code>modifiers</code>	Bit flags specifying which modifier keys the user was holding down when the event occurred. For a list of possible flags, see "F_ApiDialogEvent()" in the FDK Programmer's Reference guide.

Normally, you will want to include code in the `F_ApiDialogEvent()` function to check the properties of the item specified by `itemNum`. If the dialog box includes any items that the user can double-click, you may also want to check the `FO_DialogResource` property, `FP_DoubleClick`, to determine whether the user double-clicked in the dialog box.

Your `F_ApiDialogEvent()` function can include calls to any API function. It can get and set properties of objects in the dialog box and in the FrameMaker product session. It can also call functions, such as `F_ApiOpen()` or `F_ApiUpdateXrefs()`, to execute FrameMaker product operations.

For example, the following code handles events for a dialog box that contains one of each type of dialog item:

```

. . .
F_ObjHandleT dlgId;
. . .
#define BUTTON_1 1
#define CHECKBOX_1 2
#define TRIBOX_1 3
#define POPUP_1 4
#define SCROLLBOX_1 5
#define IMAGE_1 6
#define EDITBOX_1 7
#define RADIOBUTTON_1 8

VoidT F_ApiDialogEvent(dlgNum, itemNum, modifiers)
    IntT dlgNum;
    IntT itemNum;
    IntT modifiers;
{
    IntT state;
    F_ObjHandleT itemId;
    StringT text;
    F_StringsT labels;

    if (F_ApiGetInt(0, dlgId, FP_DoubleClick) == True)
        F_Printf(NULL, "The user double-clicked.\n");
    itemId = F_ApiDialogItemId(dlgId, itemNum);

    switch(itemNum)
    {
        case BUTTON_1:
        case CHECKBOX_1:
        case TRIBOX_1:
        case RADIOBUTTON_1:
            state = F_ApiGetInt(dlgId, itemId, FP_State);
            switch(state)
            {
                case FV_DlgOptActive:
                    F_Printf(NULL, "%d set to on.\n", itemNum);
                    break;
                case FV_DlgOptNotActive:

```

```
        F_Printf(NULL, "%d set to off.\n", itemNum);
        break;
    case FV_DlgOptDontCare:
        F_Printf(NULL, "%d set to As Is.\n", itemNum);
        break;
    }
    break;
case IMAGE_1:
case POPUP_1:
case SCROLLBOX_1:
    state = F_ApiGetInt(dlgId, itemId, FP_State);
    labels = F_ApiGetStrings(dlgId, itemId, FP_Labels);
    F_Printf(NULL, "%s was chosen from item #d.\n",
        labels.val[state], itemNum);
    break;
case EDITBOX_1:
    text = F_ApiGetString(dlgId, itemId, FP_Text);
    F_Printf(NULL, "Text box contains text: %s.\n", text);
    break;
case FV_DlgClose:
    F_Printf(NULL, "The user closed the dialog box.\n");
    break;
default:
    break;
}
}
. . .
```

Handling user actions in a modal dialog box

The API allows you to implement two different types of modal dialog boxes:

- Single-interaction dialog boxes, which close as soon as the user clicks an item
- Multiple-interaction dialog boxes, which allow the user to manipulate one or more items without closing

For example, alert boxes are single-interaction modal dialog boxes. The API dialog boxes displayed by `F_ApiPromptInt()`, `F_ApiPromptMetric()`, and `F_ApiPromptString()` are multiple-interaction modal dialog boxes.

The following sections describe how to handle user actions in each type of modal dialog box.

Handling user actions in single-interaction dialog boxes

To implement a dialog box as a single-interaction modal dialog box, follow these steps:

- 1 *Display the dialog box by calling `F_ApiModalDialog()` with `dlgNum` set to 0.*

`F_ApiModalDialog()` does not return until the user clicks a dialog item or uses another command, such as Esc or Control-c, to close the dialog box.

- 2 *After `F_ApiModalDialog()` returns, determine what the user did by getting the properties of items in the dialog box.*

For example, the following code handles user actions in a modal dialog box that contains Cancel, Apply, and Help buttons:

```
. . .
#define APPLY_BUTTON 1
#define CANCEL_BUTTON 2
F_ObjHandleT dlgId;

/* Open the resource and display the dialog box. */
dlgId = F_ApiOpenResource(FO_DialogResource, "singleInteract");

/* Determine what the user action was and respond to it. */
if(!F_ApiModalDialog(0, dlgId))
{
    if(F_ApiGetInt(dlgId, F_ApiDialogItemId(dlgId, APPLY_BUTTON),
        FP_State) == True)
        F_Printf(NULL, "Apply was clicked.\n");
    else if(F_ApiGetInt(dlgId, F_ApiDialogItemId(dlgId,
        CANCEL_BUTTON), FP_State) == True)
        F_Printf(NULL, "Cancel was clicked.\n");
    else
        F_Printf(NULL, "Dialog closed; nothing clicked.\n");
}
else
{
    /* User requested help; code to provide help goes here. */
}
. . .
```


Handling user actions in multiple-interaction dialog boxes

To implement a dialog box as a multiple-interaction modal dialog box, follow these steps:

- 1 *Display the dialog box by calling `F_ApiModalDialog()` with `dlgNum` set to a nonzero value.*

When there is a user action in the dialog box, the API calls your client's `F_ApiDialogEvent()` function, passing it the number you specified for `dlgNum`. Your client's `F_ApiDialogEvent()` function can handle user actions in the modal dialog box the same way it handles user actions in a modeless dialog box.

- 2 *Call `F_ApiReturnValue(FR_ModalStayUp)` in your client's `F_ApiDialogEvent()` callback.*

If you want to allow the user to click a button without closing the dialog box, call `F_ApiReturnValue(FR_ModalStayUp)` each time the user clicks the button.

The syntax for `F_ApiReturnValue()` is:

```
VoidT F_ApiReturnValue(IntT val);
```

This argument	Means
<code>val</code>	Specifies a return value for the current callback. To prevent a modeless dialog box from closing, set it to <code>FR_ModalStayUp</code> . For a list of the other values you can specify, see “ <code>F_ApiReturnValue()</code> ” in the FDK Programmer's Reference guide.

- 3 *To close the dialog box for an event that is not a button, call `F_ApiClose()`.*
For more information about `F_ApiClose()`, see “Closing a dialog box” on page 461.

The following code opens and displays a modal dialog box containing a checkbox. When the user clicks the checkbox, the dialog box remains on the screen. If the user clicks any other button, the dialog box closes.

```

. . .
#define DLG_NUM 1
#define CHECKBOX_NUM 14
F_ObjHandleT dlgId; /* Global variable */
. . .
/* Open resource and display dialog box. */
dlgId = F_ApiOpenResource(FO_DialogResource, "multiInteract");
F_ApiModalDialog(DLG_NUM, dlgId);
. . .
VoidT F_ApiDialogEvent(dlgNum, itemNum, modifiers)
    IntT dlgNum;
    IntT itemNum;
    IntT modifiers;
{
    /* Keeps dialog box on screen if checkbox is clicked. */
    if(itemNum == CHECKBOX_NUM)
    {
        F_Printf(NULL, "User toggled checkbox.\n");
        F_ApiReturnValue(FR_ModalStayUp);
    }
}
. . .

```

Handling user actions in multiple modeless dialog boxes

The API allows you to have multiple modeless dialog boxes open at the same time. To handle user actions in multiple dialog boxes, you must keep track of each dialog resource's number and ID. The API does not pass a dialog resource's ID to `F_ApiDialogEvent()`, so you must store each dialog resource's ID to a global variable that you associate with the dialog resource's number.

For example, the following code opens two dialog boxes and handles user actions in them:

```
. . .
#define DLG1_NUM 1
#define DLG2_NUM 2
F_ObjHandleT dlg1Id, dlg2Id; /* Global variables */
. . .
dlg1Id = F_ApiOpenResource(FO_DialogResource, "dialog1");
dlg2Id = F_ApiOpenResource(FO_DialogResource, "dialog2");
F_ApiModelessDialog(DLG1_NUM, dlg1Id);
F_ApiModelessDialog(DLG2_NUM, dlg2Id);
. . .
VoidT F_ApiDialogEvent(dlgNum, itemNum, modifiers)
    IntT dlgNum;
    IntT itemNum;
    IntT modifiers;
{
    F_ObjHandleT itemId;
    if (itemNum == FV_DlgClose) return;

    switch(dlgNum)
    {
        case DLG1_NUM:
            itemId = F_ApiDialogItemId(dlg1Id, itemNum);
            /* Code to get item properties goes here. */
            break;
        case DLG2_NUM:
            itemId = F_ApiDialogItemId(dlg2Id, itemNum);
            /* Code to get item properties goes here. */
            break;
    }
}
. . .
```

Handling special events in a modeless dialog box

There are several special dialog box events that don't apply to specific dialog items. If one of these events occurs, the API sets the `itemNum` parameter of the `F_ApiDialogEvent()` function to one of the following negative integer constants:

Constant	Event
<code>FV_DlgClose</code>	<p>The dialog box closed. A dialog box closes when the user makes a dialog close gesture (such as pressing Control-c), when the user exits the FrameMaker product. It also closes the dialog box when your client specifies the dialog box ID in a call to <code>F_ApiClose()</code>, <i>and</i> your code does not call <code>F_ApiReturnValue()</code> to set a return value of <code>FR_ModalStayUp</code>.</p> <p>The API does not specify how a dialog box is closed; it sets <code>itemNum</code> to <code>FV_DlgClose</code> regardless of how the dialog box was closed.</p> <p>Note: A dialog can also be closed due to workspace-related operations, such as switching workspaces. In such cases, instead of the dialog event <code>FV_DlgClose</code>, the notification <code>FA_Note_QuitModelessDialog</code> is sent to the client.</p>
<code>FV_DlgEnter</code>	The user moved input focus to the dialog box.
<code>FV_DlgNeedsUpdate</code>	<p>This dialog event should be issued if, within a workspace, the dialog gets visible from a minimized/iconic state. This event indicates that the client's modeless dialog has become visible and should be updated, so that it does not display stale information.</p>
<code>FV_DlgNoChange</code>	The user pressed Shift-F8 to set all the items in a dialog box to their As Is states.
<code>FV_DlgReset</code>	The user pressed Shift-F9 to reset the items in the dialog box to the values they had the last time the user clicked Apply.
<code>FV_DlgUndo</code>	The user chose Undo (Control-z).

Some FrameMaker product dialog boxes, such as the Paragraph Designer and the Character Designer, support the `FV_DlgNoChange` and `FV_DlgReset` events. If your client uses dialog boxes similar to these dialog boxes, it should include code to handle these events. For example, to make your client support the `FV_DlgNoChange` event, add code to do the following to a dialog box:

- Set the state of every tribox to `FV_DlgOptDontCare`.
- Set the text of every text box to an empty string.
- Set the state of any pop-up menus that contain an As Is item to the index of that item.

Closing a dialog box

To close a dialog box, call `F_ApiClose()`. The syntax for `F_ApiClose()` is:

```
F_ObjHandleT F_ApiClose(F_ObjHandleT objId,  
                        IntT flags);
```

This argument	Means
<code>objId</code>	The ID of the dialog box to close.
<code>flags</code>	Currently an unused parameter. Set it to 0.

You can close a dialog box anywhere in your client code, including the `F_ApiDialogEvent()` callback. For example, you can call `F_ApiClose()` to close the dialog box after a dialog event for clicking a radio button.

However, you can call `F_ApiReturnValue()` to set `FR_ModalStayUp` in the callback for a given dialog box event. In that case, the `FR_ModalStayUp` overrides any call to `F_ApiClose()` for the same event.

.....
IMPORTANT: Check the ID you pass to `F_ApiClose()` to make sure it is not 0. If you call `F_ApiClose()` with `objId` set to 0, it quits the Frame session, abandoning any unsaved changes.
.....

For example, the following code closes a dialog box:

```
. . .  
F_ObjHandleT dlgId;  
. . .  
if(dlgId != 0) F_ApiClose(dlgId, 0);  
. . .
```


Using Imported Files and Insets

.....

⋮

This chapter provides instructions for using imported files and insets in your client. It discusses the types of imported files and insets and describes how to import files. It describes how to write a filter client, a client that translates Frame files to or from other file formats. It also discusses *graphic inset editors*, clients that save graphics in a format that FrameMaker can import.

Types of imported files and insets

FrameMaker products and the Frame API allow you to import graphic and text files by copy and by reference. The following sections briefly describe the types of imported files and insets.

Imported text files

When the user imports a text file by copy, the FrameMaker product copies the file's text into the FrameMaker product document. The FrameMaker product no longer needs the original file to display the text.

When the user imports a text file by reference, the FrameMaker product creates an object called a *text inset*. A text inset contains a locked copy of the imported text. It also references the imported text file and specifies how the text is displayed in the FrameMaker product document. The FrameMaker product uses the information in a text inset to display the inset's text. Each time it updates a text inset, the FrameMaker product uses the text in the referenced file to replace the text in the inset.

There are several types of text insets, which correspond to the types of text files you can import. The following table lists the text inset file types and the corresponding API text inset objects.

File type	API inset object that represents it
Text	FO_TiText
	FO_TiTextTable
Frame binary document	FO_TiFlow
MIF	FO_TiFlow

Client text insets

The API allows you to create a special type of text inset called a *client text inset*. The text for a client text inset is not directly provided by an external file. Instead, it is provided and maintained by an FDK client. The API represents each client text inset in a document with an `FO_TiApiClient` object. For more information on client text insets, see “Client text insets” on page 471.

Imported graphics files

When the user imports a graphics file, the FrameMaker product creates an object called a *graphic inset*. The API uses an `FO_Inset` object to represent each graphic inset in a document. An `FO_Inset` object has properties that specify aspects of how an imported graphic appears, such as its size and scaling.

If the user imports a graphics file by copy, the resulting graphic inset is called an *internal graphic inset*. An internal graphic inset contains all of an imported graphic’s data. If the user imports a graphics file by reference, the resulting inset is called an *external graphic inset*. An external graphic inset does not contain all the data for an imported graphic. Instead, its `FP_InsetFile` property specifies an external file, which contains the data the FrameMaker product uses to display the graphic.

For a more detailed description of graphic insets and how to manipulate them, see “Graphic inset properties” on page 493.

Importing text and graphics

To import text or graphics into a FrameMaker product document, use `F_ApiImport()`. With `F_ApiImport()`, you can specify aspects of the Import operation, such as whether to import a file by reference or by copy.

The syntax for `F_ApiImport()` is:

```
F_ObjHandleT F_ApiImport(F_ObjHandleT enclosingDocId,
    F_TextLocT *textLocP
    StringT filename,
    F_PropValsT *importParamsp,
    F_PropValsT **importReturnParamssp);
```

This argument	Means
<code>enclosingDocId</code>	The ID of the document into which to import the file.
<code>textLocP</code>	The text location at which to import the file.
<code>filename</code>	The full pathname of the file to import. For information on how to specify pathnames on different platforms, see the <i>FDK Platform Guide</i> for your platform.
<code>importParamsp</code>	A property list telling the FrameMaker product how to import the file and how to respond to errors and other conditions. To use the default list, specify <code>NULL</code> .
<code>importReturnParamssp</code>	A property list that provides information about how the FrameMaker product imported the file. It must be initialized before you call <code>F_ApiImport()</code> . For a list of properties in this property list, see “ <code>F_ApiImport()</code> ” in the <i>FDK Programmer’s Reference guide</i> .

.....
IMPORTANT: *Always initialize the pointer to the property list that you specify for `importReturnParamssp` to `NULL` before you call `F_ApiImport()`.*

If you import a text file by reference, `F_ApiImport()` creates a text inset and returns its ID. Otherwise, it returns `0`.

The steps for using `F_ApiImport()` are similar to the steps for calling `F_ApiOpen()` and `F_ApiSave()`. To call `F_ApiImport()`, do the following:

- 1 *Initialize the pointer to the `importReturnParamssp` property list to `NULL`.*
- 2 *Create an `importParamsp` property list.*
 You can get a default list by calling `F_ApiGetImportDefaultParams()`, or you can create a list from scratch. For a description of the default list returned by `F_ApiGetImportDefaultParams()`, see “`F_ApiGetImportDefaultParams()`” in the *FDK Programmer’s Reference guide*. For information on creating a property list from scratch, see “Creating an `openParamsp` script from scratch” on page 241 in this manual.
- 3 *Call `F_ApiImport()`.*

4 *Check the Import status.*

Use `F_ApiCheckStatus()` to check the returned values in the `importReturnParamspp` list for information about how the FrameMaker product imported the file.

5 *Deallocate memory for the `importParamsp` and `importReturnParamspp` property lists.*

Use `F_ApiDeallocatePropVals()` to deallocate memory for the lists.

The following sections provide examples of how to import several specific types of files.

Importing the main flow of a Frame document file

The following code imports the main flow of a Frame document file by reference. It uses the formatting from the source document for the imported text. If the imported file isn't a FrameMaker product document file, it displays an alert.

```
. . .
F_PropValsT params, *returnParamsp = NULL;
F_ObjHandleT docId;
F_TextRangeT tr;
IntT i;

/* Get default import list. Return if it can't be allocated. */
params = F_ApiGetImportDefaultParams();
if(params.len == 0) return;

/* Get current selection. Return if there isn't one. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tr = F_ApiGetTextRange(FV_SessionId, docId, FP_TextSelection);
if(tr.beg.objId == 0) return;

/* Change property to use formatting from source document. */
i = F_ApiGetPropIndex(&params, FS_FormatImportedText);
params.val[i].propVal.u.ival = FV_SourceDoc;

F_ApiImport(docId, &tr.beg, "/tmp/frame.doc",
            &params, &returnParamsp);

if (!F_ApiCheckStatus(returnParamsp, FV_ImportedMakerDoc))
    F_ApiAlert("File wasn't a Frame document.",
              FF_ALERT_CONTINUE_NOTE);

/* Deallocate property lists. */
F_ApiDeallocatePropVals(&params);
F_ApiDeallocatePropVals(returnParamsp);
. . .
```

Importing a graphic

The following code imports a graphic file by copy. It prevents the API from importing the file if it is not a graphic.

```

. . .
F_PropValsT params, *returnParamsp = NULL;
F_ObjHandleT docId;
F_TextRangeT tr;
IntT i;

/* Get default import list. Return if it can't be allocated. */
params = F_ApiGetImportDefaultParams();
if(params.len == 0) return;

/* Get current insertion point. Return if there isn't one. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tr = F_ApiGetTextRange(FV_SessionId, docId, FP_TextSelection);
if(tr.beg.objId == 0) return;

/* Change property to import by copy. */
i = F_ApiGetPropIndex(&params, FS_HowToImport);
params.val[i].propVal.u.ival = FV_DoByCopy;

/* Change properties to disallow nongraphic files. */
i = F_ApiGetPropIndex(&params, FS_DisallowDoc);
params.val[i].propVal.u.ival = True;
i = F_ApiGetPropIndex(&params, FS_DisallowMIF);
params.val[i].propVal.u.ival = True;
i = F_ApiGetPropIndex(&params, FS_DisallowPlainText);
params.val[i].propVal.u.ival = True;

F_ApiImport(docId, &tr.beg, "/tmp/agraphic.xwd",
            &params, &returnParamsp);

if (F_ApiCheckStatus(returnParamsp, FV_BadImportFileType))
    F_ApiAlert("File isn't importable.", FF_ALERT_CONTINUE_NOTE);

/* Deallocate property lists. */
F_ApiDeallocatePropVals(&params);
F_ApiDeallocatePropVals(returnParamsp);

. . .

```

Importing a text file

The following code imports a text file by reference into a table. It parses each paragraph in the text file into a row of cells, interpreting each tab in the paragraph as a cell separator. Notice how the code uses the `FS_FileTypeHint` parameter to specify the encoding for the text file.

```

. . .
F_PropValsT params, *returnParamsp = NULL;
F_ObjHandleT docId;
F_TextRangeT tr;
IntT i;

/* Get default import list. Return if it can't be allocated. */
params = F_ApiGetImportDefaultParams();
if(params.len == 0) return;

/* Get current insertion point. Return if there isn't one. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tr = F_ApiGetTextRange(FV_SessionId, docId, FP_TextSelection);
if(tr.beg.objId == 0) return;

/* Change properties to import text into a table. */
i = F_ApiGetPropIndex(&params, FS_FileIsText);
params.val[i].propVal.u.ival = FV_DoImportAsTable;
/* The following specifies the file format as ISO encoded text.
*/
i = F_ApiGetPropIndex(&params, FS_FileTypeHint);
params.val[i].propVal.u.sval =
F_ApiCopyString((ConStringT)"0001PGRFTXIS");
i = F_ApiGetPropIndex(&params, FS_DisallowMIF);
params.val[i].propVal.u.ival = True;
i = F_ApiGetPropIndex(&params, FS_DisallowDoc);
params.val[i].propVal.u.ival = True;
i = F_ApiGetPropIndex(&params, FS_DisallowGraphicTypes);
params.val[i].propVal.u.ival = True;
i = F_ApiGetPropIndex(&params, FS_ImportTblTag);
params.val[i].propVal.u.sval =
    (StringT) F_StrCopyString("Format A");
i = F_ApiGetPropIndex(&params, FS_CellSeparator);
params.val[i].propVal.u.sval = (StringT) F_StrCopyString("\t");

F_ApiImport(docId, &tr.beg, "/tmp/mydata.txt",
    &params, &returnParamsp);

```

```

if (F_ApiCheckStatus(returnParamsp, FV_BadImportFileType))
    F_ApiAlert("File isn't importable.", FF_ALERT_CONTINUE_NOTE);

/* Deallocate property lists. */
F_ApiDeallocatePropVals(&params);
F_ApiDeallocatePropVals(returnParamsp);
. . .

```

Importing a page of a PDF file

An FDK client can use the API scriptable property `FS_PDFPageNum` to import a particular page of a PDF document. The following example illustrates the use of this property.

```

IntT index;

IntT pageNum;

F_PropValsTparams;

/* Get the default params list for import */
params = F_ApiGetImportDefaultParams();

/* Get the index of the FS_PDFPageNum property in this list */
index = F_ApiGetPropIndex(&params, FS_PDFPageNum);

/* Specify the page number to be imported, here page 3 of the
PDF doc*/
pageNum = 3;

params.val[index].propVal.u.ival = pageNum;

/* call F_ApiImport : See F_ApiImport documentation for
details*/

...

F_ApiImport(...);

```

Updating text insets

To update text insets, call `F_ApiUpdateTextInset()`. The syntax for `F_ApiUpdateTextInset()` is:

```
IntT F_ApiUpdateTextInset(F_ObjHandleT docId,  
                          F_ObjHandleT textInsetId);
```

This argument	Means
<code>docId</code>	The ID of the document containing the inset.
<code>textInsetId</code>	The ID of the text inset to update. To update all the insets in the specified document, specify 0.

`F_ApiUpdateTextInset()` updates a text inset only if it is stale. The FrameMaker product determines whether a text inset is stale by comparing the modification date of the inset's source file with the inset's `FP_TiLastUpdate` property. To force `F_ApiUpdateTextInset()` to update an inset, set the inset's `FP_TiLastUpdate` property to 0 before calling it. You do not need to unlock any insets when you call `F_ApiUpdateTextInset()`.

Client text insets

Client text insets allow your client to display and dynamically update segments of locked text in a Frame document. The following sections describe how to create and update client text insets.

Creating a client text inset

To create a client text inset, use `F_ApiNewAnchoredObject()`. To add text to it, use `F_ApiAddText()`. After you create an inset, you may also want to set the properties listed in the following table to provide information about the inset to the user and the FrameMaker product.

Property	Type	Meaning
<code>FP_TiClientName</code>	<code>StringT</code>	The registered name of your client.
<code>FP_TiClientSource</code>	<code>StringT</code>	The name that appears as the source in the Text Inset Properties dialog box.
<code>FP_TiClientType</code>	<code>StringT</code>	The name that appears as the source type in the Text Inset Properties dialog box.
<code>FP_Name</code>	<code>StringT</code>	The inset name. It is not automatically assigned by the FrameMaker product.

You can also use a client text inset's `FP_TiClientData` property to store data, such as an SQL query string, which your client can use to update the inset.

For example, the following code creates a client text inset containing the text `Inset text`:

```

. . .
F_ObjHandleT docId, insetId;
F_TextRangeT tr;

/* Add the inset at the current insertion point. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tr = F_ApiGetTextRange(0, docId, FP_TextSelection);
if(tr.beg.objId == 0) return;
insetId = F_ApiNewAnchoredObject(docId, FO_TiApiClient,
                                &tr.beg);
tr.beg.offset++; /* Increment textLoc so it's in the inset. */

/* Unlock the inset, add text to it, and then relock it. */
F_ApiSetInt(docId, insetId, FP_TiLocked, False);
F_ApiAddText(docId, &tr.beg, "Inset text");
F_ApiSetInt(docId, insetId, FP_TiLocked, True);

/* Set some other inset info.*/
F_ApiSetString(docId, insetId, FP_TiClientData,
               "Select..Where...");
F_ApiSetString(docId, insetId, FP_TiClientSource,
               "Larry's SQL Client");
F_ApiSetString(docId, insetId, FP_TiClientType,
               "SQL Query");
. . .

```

Updating a client text inset

The API issues the following notifications when client text insets need to be updated:

- `FA_Note_UpdateAllClientTi`, when the user or an FDK client has instructed the FrameMaker product to update all the insets in the document
- `FA_Note_UpdateClientTi`, when the user or an FDK client has instructed the FrameMaker product to update one of your client's insets

When it issues these notifications, the API sets the `docId` parameter of your client's `F_ApiNotify()` callback to the ID of the enclosing document.

For the `FA_Note_UpdateClientTi` notification, it also sets the `iparm` parameter of your client's `F_ApiNotify()` callback to the ID of the inset. Your client can use the `FP_TiLastUpdate` property of an inset to determine whether it is stale.

To keep your client's text insets updated, you should request these notifications and include code in your client's `F_ApiNotify()` callback to handle them.

To modify a client text inset's contents, you must first unlock it by setting its `FP_TiLocked` property to `False`. When you are finished modifying its contents, you should set its `FP_TiLocked` property back to `True`.

To update a client text inset, you may want to delete its current contents. The API provides a function, `F_ApiDeleteTextInsetContents()`, which makes this easier. The syntax for `F_ApiDeleteTextInsetContents()` is:

```
IntT F_ApiDeleteTextInsetContents (F_ObjHandleT docId,
    F_ObjHandleT insetId);
```

This argument	Means
<code>docId</code>	The ID of the document containing the text inset
<code>insetId</code>	The text inset containing the text to be deleted

If your client is unable to update one of its client text insets, it should set the inset's `FP_TiIsUnresolved` property to `True`.

For example, the following code handles the `FA_Note_UpdateClientTi` notification:

```

. . .
/* Request notification. */
F_ApiNotification(FA_Note_UpdateClientTi, True);

/* F_ApiNotify() function to handle notifications. */
VoidT F_ApiNotify(notification, docId, sparm, iparm)
    IntT notification;
    F_ObjHandleT docId;
    StringT sparm;
    IntT iparm;
{
    F_TextRangeT tr;
    if(notification == FA_Note_UpdateClientTi)
    {
        FA_errno = FE_Success; /* Initialize and check later. */
        tr = F_ApiGetTextRange(docId, iparm, FP_TextRange);

        /* Unlock inset so it can be modified.*/
        F_ApiSetInt(docId, iparm, FP_TiLocked, False);

        /* Delete existing contents and add some new stuff.*/
        F_ApiDeleteTextInsetContents(docId, iparm);
        F_ApiAddText(docId, &tr.beg, "New text");

        /* If there were errors, the inset is unresolved. */
        if(FA_errno != FE_Success)
            F_ApiSetInt(docId, iparm, FP_TiIsUnresolved, True);

        /* Relock inset.*/
        F_ApiSetInt(docId, iparm, FP_TiLocked, True);
    }
}
. . .

```

Displaying a Text Inset Properties dialog box

When a user double-clicks a text inset that isn't a client text inset, the FrameMaker product displays the Text Inset Properties dialog box. This dialog box provides information about the text inset, such as the last modification date of the inset's source file and the date the inset was last updated. It also provides buttons for the user to execute some operations, such as convert the inset to text.

When a user double-clicks one of your client's text insets, the FrameMaker product does not display a Text Inset Properties dialog box. To display a Text Inset Properties dialog box, your client must request the `FA_Note_DisplayClientTiDialog` notification. If your client has requested this notification, when the user double-clicks one of its insets, the FrameMaker product calls the client's `F_ApiNotify()` callback with notification set to `FA_Note_DisplayClientTiDialog` and `iparm` set to the inset ID. When your client receives the notification, it can display its own Text Inset Properties dialog box. This dialog box does not need to appear the same as the FrameMaker product Text Inset Properties dialog box. For example, if your client updates client text insets by executing database queries, its Text Inset Properties dialog box could provide a text field for the user to enter a new query.

After your client displays its Text Inset Properties, it should call `F_ApiReturnValue()` with `retVal` set to `FR_DisplayedTiDialog`. This notifies the FrameMaker product that the dialog box has been displayed.

For more information on requesting and responding to notifications, see “Responding to user-initiated events or FrameMaker product operations” on page 219. For more information on using custom dialog boxes in your client, see Chapter 11, “Handling Custom Dialog Box Events.”

Writing filter clients

You can use the FDK to create filter clients that translate Frame files to or from other file formats. The FrameMaker product calls an import filter client when the user or another client attempts to open or import a file with a specified format. It calls an export filter client when the user chooses a particular format from the Format pop-up menu of the Save As dialog box or the user or another client saves a file with a specified suffix.

Filter clients that filter text file formats are called *text filter clients*. Filter clients that filter graphic file formats are called *graphic filter clients*. The following sections describe how to write each type of filter client and how to register filter clients.

Writing text import filters

The FrameMaker product invokes a text import filter in the following situations:

- The user attempts to open a file with a format that the client filters.
- The user attempts to import a file with a format that the client filters.
- Another client attempts to import or open a file with a format that the client filters.
- The FrameMaker product attempts to update a text inset that references a file with a format that the client filters.

The FrameMaker product invokes the client the same way in each of these situations. It calls the client's `F_ApiNotify()` callback with `notification` set to `FA_Note_FilterIn`, `docId` set to the ID of the active document (if there is one), and `sparm` set to the pathname of the file to filter.

The client's `F_ApiNotify()` callback should do the following to respond to the FrameMaker product's call:

1 *Create a new, invisible FrameMaker product document.*

The client can create the document with `F_ApiOpen()` or `F_ApiCustomDoc()`. For more information on using these functions, see "Creating documents" on page 246. If the new document is not invisible, it will cause an error.

2 *Filter the contents of the specified file into the new document.*

The client can use Frame API calls, such as `F_ApiAddText()` and `F_ApiNewTable()`, to add content to the document. For more information on adding text and objects to a document, see Chapter 6, "Manipulating Text" and Chapter 8, "Creating and Deleting API Objects."

The FrameMaker product allows users and clients that call `F_ApiImport()` to specify into which flow of a document to import. The user or client can specify any flow in the document. A filter client should generally filter a file into the main flow of the document it creates. However, it can filter the file into any flow in the document. For more information on main flows, see "Main flows" on page 98.

3 *Call `F_ApiReturnValue()` to indicate whether the file was successfully filtered.*

If the client successfully filters the file, it should call `F_ApiReturnValue(docId)`, where `docId` is the ID of the Frame document the filter created. If the client fails, it should call `F_ApiReturnValue(0)`.

When the client's `F_ApiNotify()` callback returns, the FrameMaker product checks the value set by the `F_ApiReturnValue()` call. If the value set by the `F_ApiReturnValue()` call is 0, the FrameMaker product displays an alert notifying the user that the file could not be opened or imported.

If the value set by the `F_ApiReturnValue()` call is the ID of the new document the client created, what the FrameMaker product does depends on how the filter call was

initiated. The following table summarizes the situations in which a filter call can be initiated and what the FrameMaker product does in each situation.

How the filter call was initiated	What the FrameMaker product does
The user attempted to open the filterable file.	Makes the document created by the filter client visible.
Another client attempted to open the filterable file (with <code>F_ApiOpen()</code>).	Returns the ID of the document created by the filter client to the other client's <code>F_ApiOpen()</code> call. If the filter fails to open the filterable file, the FrameMaker product returns 0 to the other client's <code>F_ApiOpen()</code> call. If the <code>F_ApiOpen()</code> call specified a property list with <code>FS_MakeVisible</code> set to <code>True</code> , the FrameMaker product makes the document visible.
The user attempted to import the filterable file by copy into an existing document.	Copies the specified flow of the document the filter client created into the existing document. Closes the document the filter client created.
Another client attempted to import the filterable file by copy (with <code>F_ApiImport()</code>) into an existing document.	Creates a text inset in the existing document and then copies the specified flow of the document the filter client created into the text inset. Closes the document the filter client created. Sets the <code>FP_TiFile</code> property of the inset to the pathname of the filterable file. Returns the ID of the inset created by the filter client to the other client's <code>F_ApiImport()</code> call. Sets the <code>FP_ImportHint</code> property of the text inset to a string specifying the filter client. This string is based on the format ID you specified when you registered your client. For more information on format IDs, see "Specifying format IDs and filetype hint strings" on page 480. For the complete syntax of import hint strings, see "Syntax of <code>FP_ImportHint</code> strings" in the <i>FDK Programmer's Reference</i> .
The user attempted to import the filterable file by reference into an existing document.	Creates a text inset in the existing document and then copies the specified flow of the document the filter client created into the text inset. Closes the document the filter client created. Sets the <code>FP_TiFile</code> property of the inset to the pathname of the filterable file. Returns the ID of the inset created by the filter client to the other client's <code>F_ApiImport()</code> call. Sets the <code>FP_ImportHint</code> property of the text inset to a string specifying the filter client. This string is based on the format ID you specified when you registered your client. For more information on format IDs, see "Specifying format IDs and filetype hint strings" on page 480. For the complete syntax of import hint strings, see "Syntax of <code>FP_ImportHint</code> strings" in the <i>FDK Programmer's Reference</i> .
Another client attempted to import the filterable file by reference (with <code>F_ApiImport()</code>) into an existing document.	Creates a text inset in the existing document and then copies the specified flow of the document the filter client created into the text inset. Closes the document the filter client created. Sets the <code>FP_TiFile</code> property of the inset to the pathname of the filterable file. Returns the ID of the inset created by the filter client to the other client's <code>F_ApiImport()</code> call. Sets the <code>FP_ImportHint</code> property of the text inset to a string specifying the filter client. This string is based on the format ID you specified when you registered your client. For more information on format IDs, see "Specifying format IDs and filetype hint strings" on page 480. For the complete syntax of import hint strings, see "Syntax of <code>FP_ImportHint</code> strings" in the <i>FDK Programmer's Reference</i> .
The FrameMaker product attempted to update a text inset that references the filterable file.	Replaces the contents of the inset with the specified flow of the document the filter client created. Closes the document the filter client created.

For a simple example of a text import filter, see "A simple FDE filter" on page 538.

Writing graphic import filters

The FrameMaker product invokes a graphic import filter in the following situations:

- When the user attempts to open a graphic file with a format that the client filters
- When the user attempts to import a graphic file with a format that the client filters
- When another client attempts to import or open a graphic file with a format that the client filters

The FrameMaker product invokes the client the same way in each of these situations. It calls the client's `F_ApiNotify()` callback with `notification` set to `FA_Note_FilterIn`, `docId` set to the ID of the active document (if there is one), and `sparm` set to the pathname of the file to filter.

The client's `F_ApiNotify()` callback should do the following to respond to the FrameMaker product's call:

1 Determine whether to open or import the graphic file.

If the user or another client is attempting to open the graphic file, the FrameMaker product sets the `docId` parameter to 0 when it calls the client's `F_ApiNotify()` callback. If the user or client is attempting to import the graphic file into an existing document, the FrameMaker product sets the `docId` parameter to a document ID.

2 To open the graphic file, create a new FrameMaker product document.

The client can create the document with `F_ApiOpen()` or `F_ApiCustomDoc()`. The client can filter the imported graphic directly onto a page of the document or it can create an anchored frame for the graphic.

3 To import the graphic file into an existing document, determine where to import the graphic.

The client should check the document specified by the `docId` parameter of the `F_ApiNotify()` call. If the document has an insertion point, the client should create an anchored frame at the insertion point to filter the graphic into. If there is a selected frame in the document, the client should filter the graphic into the frame.

4 Filter the graphic file into the FrameMaker product document.

The client can translate the graphic file into FrameMaker product graphic objects such as ellipses, lines, and rectangles (`FO_Ellipse`, `FO_Line`, and `FO_Rectangle` objects). For more information on creating FrameMaker product graphic objects, see "Creating graphic objects" on page 365.

If the graphic file contains a complex graphic, the client can create a graphic inset. Graphic insets provide graphic data in standard formats, such as TIFF or `FrameVector`, which the FrameMaker product can use to display and print a graphic. For more information on creating graphic insets, see "Graphic inset properties" on page 493.

Note that a client does not need to be a graphic inset editor to create a graphic inset. A filter client can also create a graphic inset.

Writing export filters

The FrameMaker product invokes an export filter when the user chooses a particular format from the Format pop-up menu of the Save As dialog box or when the user or another client saves a file with a specified suffix. The FrameMaker product calls the client's `F_ApiNotify()` callback with `notification` set to `FA_Note_FilterOut`, `docId` set to the ID of the document to filter, and `sparm` set to the pathname of the file to filter the document into. The client's `F_ApiNotify()` callback should create the specified file if it does not already exist and filter the contents of the FrameMaker product document into it.

Registering filters

For the FrameMaker product to call your client to filter files, you must register the client and the formats it filters. For this, you specify the filename extensions of the formats your client filters in the client's entry in the registration file.

To identify your filter to the FrameMaker product, you specify a vendor ID and format ID when you register it. The format ID is a four-character string you choose to identify the format on all platforms. The vendor ID is a four-character string that identifies the filter vendor. The FrameMaker product uses these IDs to identify your filter when it reimports a file imported by reference.

Specifying format IDs and filetype hint strings

When you register a filter, the FrameMaker product uses the information you supply to associate that filter with a specific file format. The product also uses that information to associate a filter with an imported graphic or a text inset. Internally, this information is stored in a *filetype hint*. The filetype hint includes the filter version, the vendor, and the file format this filter handles.

Assume you create a filter client that translates Himyaritic documents to English, and you assign it the format ID `'HIM '`. When you import a Himyaritic file by reference into a FrameMaker product document, the FrameMaker product creates a text inset and saves the format ID in the inset's import hint (`FP_ImportHint`) property. The next time you open the document, the FrameMaker product uses the import hint to update the text inset. It looks for a client with the format ID `'HIM '`. If it finds a client with this ID, it uses that client to update the inset. Note that this mechanism works across platforms. If you move the document and text file to another platform, the hint string will still indicate the correct filter, assuming the filter is installed on the new platform.

Specifying format IDs

You can make up format IDs for specific file formats. However, FrameMaker products reserve the following format IDs for the specified file formats.

Value	Description
`CDR`	Corel Draw
`CGM`	Computer Graphics Metafile
`CVBN`	Corel Ventura compound document
`DIB`	Device-independent bitmap
`DRW`	Micrografx CAD
`DXF`	Autodesk CAD
`DWG`	Autodesk “Drawing” format
`EMF`	Enhanced Metafile
`EPS`	Encapsulated PostScript
`EPSB`	Encapsulated PostScript Binary
`EPSD`	Encapsulated PostScript (DCS)
`EPSF`	Encapsulated PostScript
`EPSI`	Encapsulated PostScript Interchange
`FRMV`	FrameVector
`G4IM`	CCITT Group 4 to Image
`GEM`	GEM (Windows)
`GIF`	Graphic Image File Format (CompuServe)
`HTML`	Hyper Text Markup Language
`HPGL`	Hewlett-Packard Graphics Language
`IAF`	Interleaf compound document
`IGES`	CAD format
`JAW`	ICH15
`JBW`	ICH16
`J2C`	JPEG 2000 with extension J2C
`J2K`	JPEG 2000 with extension J2K

Value	Description
'JP2'	JPEG 2000 with extension JP2
'JPC'	JPEG 2000 with extension JPC
'JPF'	JPEG 2000 with extension JPF
'JPX'	JPEG 2000 with extension JPX
'JPEG'	JPEG
'MOOV'	QuickTime Movie
'MIAF'	MIF to IAF export
'MIF'	Adobe Maker Interchange Format
'MML'	Maker Markup Language (MML)
'MRTF'	MIF to RTF export
'MWPB'	MIF to WordPerfect export
'OLE'	Object Linking and Embedding Client (Microsoft)
'P65'	Pagemaker 7.0 document
'PCX'	PC Paintbrush
'PMD'	Pagemaker 7.0 document
'PMT'	Pagemaker 7.0 template
'PNG'	Portable Network Graphics
'PDF'	Portable Document Format
'PICT'	QuickDraw PICT
'PNTG'	MacPaint
'PSD'	Adobe Photoshop
'QXD'	Quark
'RGB'	RGB
'RTF'	Microsoft's RTF compound document
'SGML'	Standard Generalized Markup Language
'SNRF'	Sun Raster File
'SVG'	Scalable Vector Graphics
'SVPD'	SVG to PDF

Value	Description
`SWF`	Adobe Flash File
`T65`	Pagemaker 7.0 document
`TANS`	ANSI Text Encoding
`TBG5`	Big5 format
`TEUH`	UCCNS format
`TEXT`	Text
`TIFF`	Tag Image File Format
`TJIS`	JIS (Japanese Industrial Standards) text encoding
`TSJS`	Shift JIS text encoding
`TKOR`	Korean text encoding format
`TMAC`	Max Text Format
`TRFA`	TRFA
`TRFE`	TRFE
`TRFS`	TRFS
`TU1B`	Unicode UTF-16BE
`TU1L`	Unicode UTF-16LE
`TU3B`	Unicode UTF-32BE
`TU3L`	Unicode UTF-32LE
`TUT8`	Unicode UTF-8
`TXHZ`	THZ format
`TXIS`	ISO text encoding
`TXGB`	Chinese National Standard format

Value	Description
<code>`TXRM`</code>	TXRM
<code>`WORD`</code>	Microsoft Word 2000
<code>`WDBN`</code>	MS Word compound document
<code>`WPBN`</code>	WordPerfect compound document
<code>`WPG`</code>	WordPerfect Graphics
<code>`XBM`</code>	X Windows bitmap
<code>`XLS`</code>	Microsoft Excel 2000
<code>`XML`</code>	eXensible Markup Language
<code>`XSLF`</code>	ADBI_XSLFO_HINT
<code>`XWD`</code>	X Windows bitmap

FrameMaker products do not supply filters for all of these formats on all platforms. However, you should not use one of these format IDs unless your client filters the corresponding file format.

For the complete syntax of import hint strings, see “Syntax of FP_ImportHint strings” in the FDK Programmer’s Reference.

Filetype hint string syntax

The FrameMaker product uses filetype hint strings for both graphic and document or text files. The hint strings are stored with imported graphics and with text insets. You also use hint strings to invoke specific filters from within your API clients. For example, to save a FrameMaker document as HTML, use the following code to specify the HTML hint string:

```

IntT i;
F_PropValsT params;
. . .
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
params = F_ApiGetSaveDefaultParams();
i = F_ApiGetPropIndex(&params, FS_FileType)
params.val[i].propVal.u.ival = FV_SaveFmtFilter;
i = F_ApiGetPropIndex(&params, FS_SaveFileTypeHint)
params.val[i].propVal.u.sval =
    F_StrCopyString((StringT)"0001ADBEHTML");
. . .
    
```

The syntax of a hint string is:

record_vers vendor format_id filter_vers filter_name

Of these, *record_vers vendor format_id* are the minimum required to specify a filter. Each field must specify a four-byte code, except for *filter_name* which can be up to 31 characters long. If necessary, you must pad out each field so that it has exactly 4 Alphanumeric characters. For example, the hint for version 1 of the XML filter must be specified as follows; 0001ADBEXML 0001. Note the space padding after the three-character code for XML. However, the last field in a hint string does not need padding. In the above example, if you did not specify the version number of the XML filter, XML would have been the last field, and it would not have required padding.

record_vers specifies the version of the record, currently 0001.

vendor is a code specifying the filter's vendor. The code is a string of four characters. The following table lists the possible codes.

Code	Meaning
ADBE	Adobe external filter
ADBI	Adobe internal filter
AIDE	Aide filter
APSL	Adobe
AW4W	Master soft filter
FRAM	FrameMaker internal filter
IDNT	Ident filter

Code	Meaning
IMAG	Imagemark filter
IVY	Ivy Filter
PGRF	Built-in Frame filters
FAPI	External Frame FDK client filter
FFLT	External Frame filters
IMAG	External ImageMark filters
XTND	External XTND filters

This is not a comprehensive list of codes. Codes may be added to this list by Frame or by developers at your site.

format_id is a code specifying the format that the filter translates. The code is a string of four characters. See “Specifying format IDs” on page 481 for information about format codes and a table that lists some of the possible codes.

filter_vers is a string of four characters identifying the version of the filter on that platform. For example, version 1.0 of a filter is represented by the string 1.0.

filter_name is a text string (up to 31 characters long) that describes the filter.

Hint strings for the standard installation of filters

The following tables list the minimal hint strings for all the import and export filters that ship with FrameMaker, and for importing and exporting text files. Your installation may not include all of these filters, and you may have other filters installed. For this reason, a code example that generates these lists follows the tables.

Hint strings

The following tables list hint strings for Windows filters:

Hint strings for import filters:

To use this import filter:	Use this hint string:
Corel Draw	0001IMAGCDR
DIB	0001FRAMDIB
Micrografx Drawing	0001IMAGDRW
AutoCAD (DWG)	0001IMAGDWG

To use this import filter:	Use this hint string:
AutoCAD (dxf)	0001IMAGDXF
Windows Enhanced Metafile	0001FRAMEMF
EPS/EPSF	0001FRAMEPSF
PDF	0001FRAMPDF
Frame Image	0001FRAMFRMI
FrameVector	0001FRAMFRMV
CCITTG4	0001IMAGG4IM
GEM	0001IMAGGEM
GIF	0001AIDEGIF WIN3
HPGL	0001IMAGHPGL
IGES	0001IMAGIGES
JPEG	0001AIDEJPEGWIN3
PCX	0001FRAMPX
QuickDraw PICT	0001IMAGPICT
Portable Network Graphics	0001IMAGPNG
MacPaint	0001FRAMPNTG
Sun Raster File	0001FRAMSNRF
TIFF	0001FRAMTIFF
Windows Metafile	0001FRAMWMF
Windows Metafile to FrameVector	0001IMAGWMF
WordPerfect Graphics	0001IMAGWPG
XWD	0001FRAMXWD
MIF	0001FRAMMIF
MML	0001FRAMMML
Text	0001FRAMTEXT
Ventura Publisher	0001FFLTCVBN
Microsoft Word for Windows 6.0/7.0	0001AW4W0490
Microsoft Word for Windows 2.0	0001AW4W0441
Microsoft Word for Windows 1.0	0001AW4W0440

Using Imported Files and Insets

Specifying format IDs and filetype hint strings

To use this import filter:	Use this hint string:
Microsoft Word Macintosh 6.0	0001AW4W049m
Microsoft Word DOS 5.0/6.0	0001AW4W0052
Microsoft Word DOS 4.0	0001AW4W0051
Microsoft Word DOS 3.0, 3.1	0001AW4W0050
Microsoft Word Mac 5.x	0001AW4W0542
Microsoft Word Mac 4.x	0001AW4W0541
Microsoft Word Mac 3.x	0001AW4W0540
Microsoft RTF	0001AW4W0191
WordPerfect DOS/Win 7.0	0001AW4W0482
WordPerfect DOS/Win 6.1	0001AW4W0481
WordPerfect DOS/Win 6.0	0001AW4W0480
WordPerfect DOS/Win 5.1	0001AW4W0071
WordPerfect DOS/Win 5.0	0001AW4W0070
WordPerfect Mac 3.0-3.5	0001AW4W0601
WordPerfect Mac 2.0-2.1	0001AW4W0600
WordPerfect Mac 1.0	0001AW4W0590
InterLeaf ASCII Format	0001AW4W0460
DCA Revisable Form Text	0001AW4W0150
DCA RFT (DisplayWrite 5)	0001AW4W0151
Ami Professional 2-3.1	0001AW4W0331
Ami Professional 1	0001AW4W0330
Lotus 1-2-3 5.0	0001AW4W0204
Lotus 1-2-3 4.0	0001AW4W0203
Microsoft Excel 5.0	0001AW4W0214
Microsoft Word97	0001AW4W3490
ICHITARO5	0001IVY JAW
ICHITARO6	0001IVY JBW
RTF Japanese	0001IVY RTF
PNG	0001AIDEPNG WIN3

To use this import filter:	Use this hint string:
PSD	0001APSLPSD WIN3
CGM Import IsoDraw	0001ISO CGM
U3D	0001ADBIU3D
SWF	0001ADBISWF
<i>Unicode Text Files:</i>	
UTF-8	0001PGRFTUT8
UTF-16LE	0001PGRFTU1L
UTF-16BE	0001PGRFTU1B
UTF-32LE	0001PGRFTU3L
UTF-32BE	0001PGRFTU3B
GIF	0001AIDEGIF WIN3
JPEG	0001AIDEJPEGWIN3
<i>JPEG 2000 with extension</i>	
JP2	"0001ADBIJP2 "
JPX	"0001ADBIJPX "
J2C	"0001ADBIJ2C "
J2K	"0001ADBIJ2K "
JPC	"0001ADBIJPC "
JPF	"0001ADBIJPF "
PNG	0001AIDEPNG WIN3
PSD	0001APSLPSD WIN3
Quark	"0001ADBIQXD "
<i>PageMaker:</i>	
7.0 Document	"0001ADBIPM D "
7.0 template	"0001ADBIPM T "
7.0 document	"0001ADBIP65 "
7.0 document	"0001ADBIT65 "

Hint strings for export filters:

To use this import filter:	Use this hint string:
FrameVector	0001FRAMFRMV
FrameImage	0001FRAMFRMI
IGES	0001IMAGIGES
QuickDraw PICT	0001IMAGPICT
EPS	0001IMAGEPS
TIFF	0001IMAGTIFF
DIB	0001IMAGDIB
GIF	0001IMAGGIF
CCITTG4	0001IMAGG4IM
JPEG	0001IMAGJPEG
Portable Network Graphics	0001IMAGPNG
Windows Metafile	0001IMAGWMF
Microsoft RTF	0001AW4W0192
Microsoft Word Win 6.0/7.0	0001AW4W0490
Microsoft Word Mac 6.0	0001AW4W049m
Microsoft Word Mac 5.x	0001AW4W0542
Microsoft Word Mac 4.x	0001AW4W0541
WordPerfect DOS/Win 5.1	0001AW4W0071
WordPerfect Mac 3.5	0001AW4W0602
RTF (Japanese)	0001IVY RTFJ
CGM Export IsoDraw	0001ISO CGM
HTML	0001ADBHTML
XML	0001ADBXML

Text import and export hint strings

The following table lists hint strings for importing and exporting text files:

To import or export this text:	Use this hint string:
Plain text	0001PGRFTEXT
Text ISO Latin 1	0001PGRFTXIS
Text Roman 8 (HEWLETT PACKARD UNIX)	0001PGRFTXRM
Text ANSI (Windows)	0001PGRFTANS
Text (Macintosh)	0001PGRFTMAC
Text ASCII	0001PGRFTASC
Japanese JIS	0001PGRFTJIS
Japanese Shift-JIS	0001PGRFTSJS
Japanese EUC	0001PGRFTEUJ
Traditional Chinese BIG 5	0001PGRFTBG5
Traditional Chinese EUC-CNS	0001PGRFTEUH
Simplified Chinese HZ	0001PGRFTXHZ
Simplified Chinese GB	0001PGRFTXGB
Korean	0001PGRFTKOR

To generate a list of filters for a given session, you print out the list of registered import filters and the list of registered export filters. The following example gets those lists and prints their contents out to the console:

```

. . .
F_Stringt importFilters, exportFilters;
IntT i;

F_Printf(NULL, (StringT)"\n\n ###IMPORT FILTERS###\n\n");
importFilters = F_ApiGetStrings(0, FV_SessionId,
FP_ImportFilters);
for (i=0; i < importFilters.len; i++)
    F_Printf(NULL, (StringT)"%s\n", importFilters.val[i]);

F_Printf(NULL, (StringT)"\n\n ###EXPORT FILTERS###\n\n");
exportFilters = F_ApiGetStrings(0, FV_SessionId,
FP_ExportFilters);
for (i=0; i < exportFilters.len; i++)
    F_Printf(NULL, (StringT)"%s\n", exportFilters.val[i]);

F_ApiDeallocateStrings(&importFilters);
F_ApiDeallocateStrings(&exportFilters);

```

Associating a file format with signature bytes

Some file formats have *signature bytes*. Signature bytes are a set of bytes with a unique value and location in a particular file format. FrameMaker products can use signature bytes to automatically identify a file's format. The documentation for the file format your client converts may contain information on the signature bytes for that format.

FrameMaker products allow you to associate a set of signature bytes with a specific file format. When the FrameMaker product opens a file containing the signature bytes, it assumes the file has the specified file format and calls the appropriate filter for that format. For more information on registering signature bytes, see the *FDK Platform Guide* for your platform.

Graphic inset properties

The API represents each graphic inset with an `FO_Inset` object. An `FO_Inset` object has the properties common to all graphic objects. It also has some properties that are specific to graphic insets. The following table lists some of these properties.

Property	Type	Meaning
<code>FP_InsetDpi</code>	<code>IntT</code>	Dots per inch (DPI). Indicates scale factor. It only applies to autosizing raster images.
<code>FP_InsetFile</code>	<code>StringT</code>	Platform-specific pathname if the inset is an external inset, or a null string (" ") if it is internal. The pathname can be document-relative.
<code>FP_InsetIsFixedSize</code>	<code>IntT</code>	True if <code>FP_Width</code> and <code>FP_Height</code> are used for the graphic's size. False if <code>autosize</code> is used.
<code>FP_InsetIsFlippedSideways</code>	<code>IntT</code>	True if inset is flipped sideways.

In addition to the properties listed in the table above, each `FO_Inset` object has special properties called *facets*, which contain data describing the imported graphic.

Each `FO_Inset` object must have at least one of the standard graphic inset facets listed in the following table.

Facet format

DCS Black
DCS Cyan
DCS Magenta
DCS Yellow
CGM
EPSI (Encapsulated PostScript)
FrameImage
FrameVector
GIF
MacPaint

Facet format

PCX

TIFF

XWD

DIB

EMF

OLE

WMF (Windows Metafile)

FrameImage facets follow Sun raster image format. FrameVector facets follow a TIFF format. For a complete description of these formats, see the online *MIF Reference* manual.

If an inset doesn't have one of the facet formats listed above, a FrameMaker product can use a filter to convert another format into FrameImage or FrameVector formats. For example, if the inset has an HPGL facet, the FrameMaker product can convert it into FrameVector format.

A FrameMaker product uses an inset's facets to display and print it. A FrameMaker product may use different facets to display and print a graphic. For example, the

When displaying an imported graphic, FrameMaker products use one of the following facet formats (in order of preference):

- Native platform format (WMF)
- FrameVector
- FrameImage and other bitmap formats
- TIFF
- Other bitmap formats

When printing an imported graphic, FrameMaker products use one of the following facet formats (in order of preference):

- EPSI (Encapsulated PostScript)
- Native platform format (WMF)
- FrameVector
- TIFF
- FrameImage and other bitmap formats

Unlike other properties, which are identified by numbers, facets are identified by names. Instead of using `propIdent.num` to identify a facet, the API sets `propIdent.num` to 0 and sets `propIdent.name` to the facet name. For more information on how the API represents property and property lists, see “Representing object characteristics with properties” on page 65.

In addition to the facets listed above, each `FO_Inset` object can have several client-specific facets. Client-specific facets contain information that your client uses. A client-specific facet can contain a complete set of data, or just a pathname to an external data file or database. You must register the names of client-specific facets with the FrameMaker product. A graphic inset can have as many client-specific facets as you want.

Facets can specify integer (`IntT`), metric (`MetricT`), or unsigned bytes (`F_UBytesT`) data. Facets, such as `EPSI` and `FrameImage`, that specify multiple characters or binary data are `F_UBytesT` facets. `F_UBytesT` is defined as:

```
typedef struct {  
    UIntT len; /* The number of unsigned bytes */  
    UByteT *val; /* The facet data */  
} F_UBytesT;
```

Internal and external graphic insets

There are two types of graphic insets: internal and external. You choose which type your client supports. Both types require a display and print facet and can have one or more client-specific facets.

Internal graphic insets

Internal graphic insets are wholly contained within a FrameMaker product document. If a graphic inset is internal, the `FO_Inset` object’s `FP_InsetFile` property is set to a null string (“”). Internal graphic insets are generally more portable than external graphic insets.

To update an internal graphic inset, the user starts the FrameMaker product and initiates an event that the graphic inset editor monitors. For example, if the user double-clicks the inset, the graphic inset editor updates the inset by setting its facets.

External graphic insets

External graphic insets are stored in an external file. The `FP_InsetFile` property of an external graphic inset is set to a platform-specific filename that specifies the file. For information on converting platform-specific pathnames to platform-independent (device-independent) pathnames, see Chapter 15, “Making I/O and Memory Calls Portable.”

Users can update external graphic insets the same way they edit internal graphic insets. Because external graphic inset data is not contained within the FrameMaker product document, users can also edit the inset with other applications besides a graphic inset editor.

To edit an external graphic inset from a FrameMaker product, the user opens the document and double-clicks the inset. The FrameMaker product launches the graphic inset editor, which updates the inset by editing the inset file.

Example

Suppose the user creates a bitmap inset with a graphic inset editor named `myeditor`. The resulting `FO_Inset` object’s property list and some of its properties are shown in Figure 12-2.

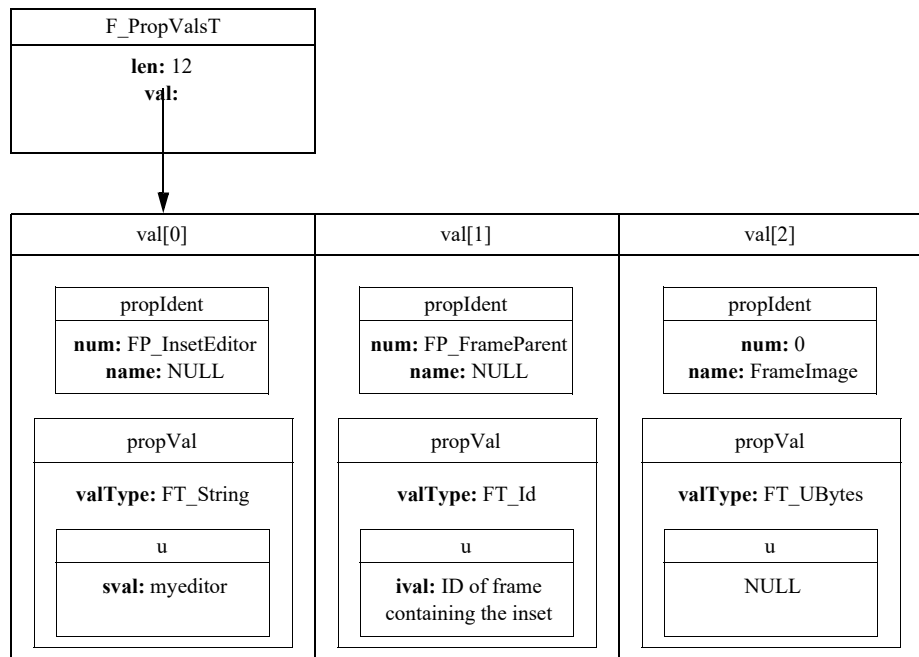


Figure 12-2 The property list for a bitmap `FO_Inset` object

The `u` union for the `FrameImage` facet is `NULL`. The `F_PropValT` structure for `F_UBytesT` facets does not provide the facet data. It only serves as a placeholder, indicating that a facet exists. To get and set the data for `F_UBytesT` facets, you must follow special procedures, which are described in “Getting and setting facets” on page 501.

Setting up your client to create and modify graphic insets

To set up your client to create and modify graphic insets, do the following:

1 Register your client as an API client.

You must register your client and all the facets it uses. For information on registering API clients, see the *FDK Platform Guide* for your platform.

2 Add an `F_ApiMessage()` callback to your client.

When the user clicks your client in the Inset Editors scroll list or double-clicks a graphic inset, the FrameMaker product calls `F_ApiMessage()` from your client. Define `F_ApiMessage()` as follows:

```
VoidT F_ApiMessage(StringT message, /* Not used for insets */
    F_ObjHandleT docId, /* ID of document containing the inset */
    F_ObjHandleT objId); /* The ID of the inset or 0 */
```

Responding to the user launching your inset editor

When the user launches your graphic inset editor by choosing it from the Inset Editors scroll list, the FrameMaker product calls the editor’s `F_ApiMessage()` callback with `objId` set to 0. You can include code in `F_ApiMessage()` that allows the user to create a graphic inset and add it to the Frame document when your editor is launched.

To insert a graphic inset, you add an `FO_AFrame` object and resize it to accommodate the inset. Then you add an `FO_Inset` object to the frame and set its properties as needed.

The following code adds an external graphic inset (described in the file `myinset.fi`) when the user chooses the editor (`myeditor`) from the Inset Editors scroll list:

```
. . .  
VoidT F_ApiMessage(message, docId, objId)  
    StringT message;  
    F_ObjHandleT docId;  
    F_ObjHandleT objId; /* 0 if launched from scroll list. */  
{  
    F_ObjHandleT afrmId, insetId;  
    F_TextRangeT tr;  
    if (!objId) {  
        /* Code that allows user to create a graphic goes here. */  
  
        /* Get the text selection and add frame at the beginning. */  
        tr = F_ApiGetTextRange(FV_SessionId, docId,  
                               FP_TextSelection);  
        afrmId = F_ApiNewAnchoredObject(docId, FO_AFrame, &tr.beg);  
  
        /* Code to resize and position frame goes here. */  
  
        /* Add the inset to the frame and set its properties. */  
        insetId = F_ApiNewGraphicObject(docId, FO_Inset, afrmId);  
        F_ApiSetString(docId, insetId, FP_InsetFile, "/tmp/myinset.fi");  
        F_ApiSetString(docId, insetId, FP_InsetEditor, "myeditor");  
    }  
}  
. . .
```

Responding to the user double-clicking a graphic inset

When the user double-clicks a graphic inset for which the `FP_InsetEditor` property specifies your graphic inset editor, the FrameMaker product calls the editor's `F_ApiMessage()` callback with `objId` set to the inset's ID. Include code in `F_ApiMessage()` that allows the user to update the inset. If the inset is an internal inset, you update it by setting its facets.

Although your API graphic inset editor should respond when the user selects it from the Inset Editors scroll list or when the user clicks a graphic inset, it can create or update insets at any time. For example, you may want to update a document's insets whenever the user opens the document or chooses a particular menu item.

If a graphic inset is an external graphic inset, you update it by editing the file specified by `FP_InsetFile`. The FrameMaker product periodically refreshes external insets against the files that describe them. If you want to ensure that the inset is refreshed

immediately, you must reset the `FO_Inset` object's `FP_InsetFile`, `FP_Height`, or `FP_Width` property.

A FrameMaker product doesn't call `F_ApiMessage()` only when the user double-clicks a graphic inset. It also calls `F_ApiMessage()` when the user clicks a hypertext marker. Therefore, make sure that the object specified by `objId` is a graphic inset and not a hypertext marker.

For example, the following code updates a graphic inset when the user clicks it:

```
. . .
VoidT F_ApiMessage(message, docId, objId)
StringT message;
F_ObjHandleT docId;
F_ObjHandleT objId;
{
if (!objId) {
    /* Code to create a new inset goes here. */
}
else{
    /* Make sure clicked object is a graphic inset. */
    if (F_ApiGetObjectype(docId,objId) == FO_Inset){
        /* Code to edit or update facets with
        * F_ApiGet[PropertyType]ByName() and
        * F_ApiSet[PropertyType]ByName() goes here.
        */
    }
}
}
. . .
```

Getting and setting graphic inset properties

To get and set individual graphic inset properties identified by property numbers, you use `F_ApiGetPropertyType()` and `F_ApiSetPropertyType()` functions just as you would with any other object properties. For example, to get and set `FP_InsetDpi`, you use `F_ApiGetInt()` and `F_ApiSetInt()`.

To get and set facets and `FO_Inset` property lists, follow the procedures discussed in the following sections.

Getting and setting facets

To get and set facets, use the following functions.

To	Use
Query an integer facet	<code>F_ApiGetIntByName()</code>
Query a metric facet	<code>F_ApiGetMetricByName()</code>
Query an <code>F_UBytes</code> facet	<code>F_ApiGetUBytesByName()</code>
Set an integer facet	<code>F_ApiSetIntByName()</code>
Set a metric facet	<code>F_ApiSetMetricByName()</code>
Set an <code>F_UBytes</code> facet	<code>F_ApiSetUBytesByName()</code>

The syntax for these functions is similar to other `F_ApiGetPropertyType()` and `F_ApiSetPropertyType()` functions, except that you must identify the property with a character string instead of an integer. For example, the syntax for `F_ApiGetUBytesByName()` is:

```
F_UBytes *F_ApiGetUBytesByName(F_ObjHandleT docId,
                               F_ObjHandleT objId,
                               StringT *propName);
```

This argument	Means
<code>docId</code>	The ID of the document containing the inset
<code>objId</code>	The ID of the inset whose facet you want to query
<code>propName</code>	The name of the facet to query

For the exact syntax of the functions that get and set facets, look up the functions in the chapter, “FDK Function Reference,” of the FDK Programmer’s Reference.

Getting and setting facets takes an additional step not needed with other properties—committing the transaction. After executing a series of gets or sets for a graphic inset’s facets, commit the transaction by getting or setting a facet named “ ”.

For example, the following code gets two facets and commits the transaction:

```
. . .
IntT myInt, err;
MetricT myMetric;
F_ObjHandleT docId, insetId;

myInt = F_ApiGetIntByName(docId, insetId, "myinteger.facet");
myMetric = F_ApiGetMetricByName(docId, insetId,
                                "mymetric.facet");
err = F_ApiGetIntByName(docId, insetId, "");
. . .
```

If you are setting facets, you commit the transaction by setting a facet named "". For example:

```
. . .
#define in (MetricT) (72 * 65536)
F_ObjHandleT docId, insetId;

F_ApiSetMetricByName(docId, insetId, "mymetric.facet", 2*in);
F_ApiSetIntByName(docId, insetId, "", 0); /* Commit */
. . .
```

To get and set `F_UByteST` facets, follow the special procedures described in the following sections.

Getting an F_UBytesT facet

Because an `F_UBytesT` facet can contain large amounts of data, it is not feasible to return all the data in a single array. Calling `F_ApiGetUBytesByName()` returns only the next chunk of a facet's data. To get all the data for an `F_UBytesT` facet, you must call `F_ApiGetUBytesByName()` repeatedly until `F_UBytesT.len` is 0. For example, the following code gets all the bytes in a facet named `my.facet`:

```
. . .
F_ObjHandleT docId, insetId;
F_UBytesT aUBytes;
do {
    aUBytes = F_ApiGetUBytesByName(docId,insetId,"my.facet");

    /* Code to do something with aUBytes goes here. */

} while (aUBytes.len)
F_ApiGetUBytesByName(docId,insetId,""); /* Commit transaction.*/
. . .
```

Setting an F_UBytesT facet

To set an `F_UBytesT` facet that contains a relatively small chunk of data, you call `F_ApiSetUBytesByName()` once. To set an `F_UBytesT` facet that contains a large chunk of data, you must call `F_ApiSetUBytesByName()` multiple times, passing a small chunk of data each time. The size of the data chunk you pass depends on the platform you are using. In general, if you use a larger size chunk, you can set the facet more quickly. However, if you use too large a size, you risk exceeding the interapplication communication mechanism's capacity.

For example, to set the EPSI facet of a graphic inset to the contents of a file named `mydata`, use the following code:

```
. . .
#include "fchannel.h"
#include "futils.h"
F_ObjHandleT docId, insetId, pageId, pFrameId;
F_UBytesT aBytes;
UByteT buf[10 * 1024];
ChannelT channel;
FilePathT *path;

/* Create inset on page frame of current page. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
pageId = F_ApiGetId(FV_SessionId, docId, FP_CurrentPage);
pFrameId = F_ApiGetId(docId, pageId, FP_PageFrame);
insetId = F_ApiNewGraphicObject(docId, FO_Inset, pFrameId);
path = F_PathNameToFilePath((StringT)"/tmp/mydata",
                             NULL, FDefaultPath);
channel = F_ChannelOpen(path, "r");
do{
    aBytes.len = F_ChannelRead(buf, 1, sizeof(buf), channel);
    aBytes.val = buf;
    F_ApiSetUBytesByName(docId, insetId, "EPSI", &aBytes);
} while (aBytes.len);

/* Commit transaction. */
F_ApiSetIntByName(docId, insetId, "", 0);
. . .
```

Getting and setting an entire FO_Inset property list

To get and set the property list for an `FO_Inset` object, you can use `F_ApiGetProps()` and `F_ApiSetProps()` as you would with any other object. However, these functions don't handle facets that contain `UBytesT` data. In the property list that `F_ApiGetProps()` returns, these facets are always `NULL`. If you construct a property list that contains `F_UBytesT` data and pass it to `F_ApiSetProps()`, `F_ApiSetProps()` sets `FA_errno` to `FE_BadPropType`. You must get and set facets that contain `UBytesT` data with `F_ApiGetUBytesByName()` and `F_ApiSetUBytesByName()`.

.....

.....

This chapter describes the support for the Unicode format.

In this chapter:

- [Introduction to Unicode Support](#)
- [Unicode Mode](#)
- [Compatibility mode](#)
- [International Components for Unicode \(ICU\)](#)
- [Mixed Mode operations](#)
- [Handling for special characters](#)

Introduction to Unicode Support

FrameMaker provide full support for the Unicode format starting version 8. The FDK too provides a *Unicode Mode* for supporting the Unicode format—the FDK APIs and the FDK functions support Unicode when running in this mode.

When FDK APIs and the FDE aren't running in Unicode Mode, they running in *Compatibility Mode*. The *Compatibility Mode* is the default mode. default mode of operation and provides backward compatibility for legacy clients.

Unicode Mode

In the *Unicode Mode*, all FDK APIs and FDE functions require string data in all input arguments to be in Unicode. All return values from (and parameters set by) FDK APIs and FDE functions in this mode will also be in Unicode. FDE string utility functions that explicitly require string encoding as a parameter or have been made for a specific encoding are an exception to these rules.

UTF Encoding for Unicode Support

Except where specifically mentioned, all APIs expect Unicode data to be in UTF-8 encoding, rather than UTF-16 or UTF-32. Providing invalid Unicode input to APIs in

Unicode Mode will cause unpredictable behavior. The API, `F_IsValidUTF8`, ascertains whether a string is valid in the UTF-8 encoding. The client must ensure that all input to an API is in the valid encoding.

Enabling Unicode Mode in the FDK

Unicode Mode must be enabled separately for the two principal parts of the FDK: APIs and the FDE.

Enabling Unicode Mode for APIs

To enable Unicode Mode for APIs, call the new API `F_ApiEnableUnicode` as follows:

```
F_ApiEnableUnicode(True);
```

Make this call before calling any other API except the APIs required for setting up communication for asynchronous clients. Any API call made before this one won't work in Unicode Mode. Calling this multiple times is safe, but only once is necessary for clients that do not unload. For clients that do unload, this must be called every time the client is reloaded.

For this reason, you should make this the first call in `F_ApiInitialize` of the client.

The callback `F_ApiInitialize` is called for all API clients including filters and document reports. Filters can enable Unicode Mode in `F_ApiInitialize` before receiving a call to the callback `F_ApiNotify` and thus receive `sparm` in Unicode.

Disabling Unicode Mode for APIs

You can subsequently disable Unicode Mode by making the following call:

```
F_ApiEnableUnicode(False);
```

Enabling Unicode Mode for the FDE

Unicode Mode for FDE functions is enabled if the encoding of the FDE has been set to UTF-8 using the function `F_FdeInitFontEncs`. The encoding, "UTF-8" is available as a possible input value for `F_FdeInitFontEncs`. This can be called as follows:

```
FontEncIdT feId;
```

```
feId = F_FdeInitFontEncs((ConStringT) "UTF-8");
```

Or, simply as:

```
F_FdeInitFontEncs((ConStringT) "UTF-8");
```

Make this call before calling any other FDE function. Any FDE function called before this won't work in Unicode Mode. It is safe to call this function more than once. Calling

`F_FdeInitFontEncs` with an encoding other than "UTF-8" disables Unicode Mode for the FDE.

Behavior of FDK APIs in Unicode Mode

In Unicode Mode, all APIs require all strings in input parameters to be valid UTF-8 strings. Strings in all return values or parameters set by APIs in this mode will be in UTF-8. For example, `F_ApiAlert` accepts UTF-8 input in Unicode Mode:

```
F_ApiEnableUnicode(True);
F_ApiAlert("This is Unicode: \xC3\xA4 \xEB\xAE\xA4 \xD8\xB4",
FF_ALERT_CONTINUE_NOTE);
```

where `0xC3 0xA4` is the UTF-8 representation of ä (Latin), `0xEB 0xAE 0xA4` of ഴ (Hangul), and `0xD8 0xB4` of ش (Arabic). Putting a `\x` followed by the hex code `C3` puts a byte `0xC3` in the string. This is a feature provided by C/C++ compilers and isn't specific to FrameMaker. This behavior might not work for some compilers. The alert produced by this call is shown below:



F_ApiGetSupportedEncodings

In Unicode Mode, the API `F_ApiGetSupportedEncodings` returns a list containing only "UTF-8" because this is the only supported encoding in this mode.

F_ApIsEncodingSupported

In Unicode Mode, `F_ApIsEncodingSupported` returns `True` only for "UTF-8" because this is the only supported encoding in this mode.

F_ApiSave

In the *Unicode Mode*, `FV_SaveFmtBinary` and `FV_SaveFmtInterchange` are equivalent to `FV_SaveFmtBinary100` and `FV_SaveFmtInterchange100` respectively when calling `F_ApiSave`. Therefore, the default mode of saving in Unicode Mode is 10 format (FM10, MIF10, FM Book 10, MIF Book 10)

F_ApiAddText F_ApiGetText, F_ApiGetText2, F_ApiGetTextForRange, F_ApiGetTextForRange2

These APIs have a peculiar behavior with respect to Symbol, Dingbats, and Webdings fonts in Unicode Mode. The internal representation of a character 'α' in Symbol ('a' with Symbol font applied) is 0x61, which is the same as the representation of 'a'. This symbol isn't stored as the Greek Unicode letter 'α', which has the Unicode code point 0x03B1. Therefore, if Symbol font is applied to 'a', it turns into 'α'. If a font like Times New Roman is subsequently applied, it turns back into 'a'. The same is true for other characters in Symbol, Dingbats, and Webdings fonts. Hence, if the text "αβχ" (which is actually the text "abc" with the Symbol font applied) is obtained using `F_ApiGetText`, the string "abc" is returned.

The symbol ∇, which has the symbol representation of 0xD1 (see the FrameMaker Character Sets document) will be represented by the byte sequence 0xC3 0x91. This byte sequence is the UTF-8 representation of the Unicode code point 0xD1, which represents the character Ñ in Unicode. Therefore, applying Symbol font to Ñ results in ∇.

Because of this representation, Symbol/Dingbats/Webdings characters that are obtained by applying the respective fonts to non-ASCII characters are returned as two-byte sequences by these APIs. For example, using `F_ApiGetText` to get the text "∇" returns the byte sequence 0xC3 0x91 (or the UTF-8 character Ñ). To convert this to a single byte symbol representation (necessary in FrameMaker 7.2 and earlier), the client must explicitly convert the UTF-8 byte sequence 0xC3 0x91 to the UTF-32 code point 0x00D1 and then store it in a single byte.

The `F_StrConvertEnc` function(s), which can also be used to convert from Symbol/Dingbats/Webdings to UTF-8, expect the input to be in Symbol/Dingbats/Webdings encodings. Therefore, the client must convert the UTF-8 byte sequence to a single byte before sending it to these functions for conversion.

The above behavior is true for other variants of `F_ApiGetText` as well. For example, to add the text ∇ in the document, the UTF-8 byte sequence 0xC3 0x91 must be sent to `F_ApiAddText`. Applying the Symbol font to this added text displays the desired character.

F_ApiGetEncodingForFont, F_ApiGetEncodingForFamily

The APIs `F_ApiGetEncodingForFont` and `F_ApiGetEncodingForFamily` have the same behavior in both *Compatibility Mode* and *Unicode Mode*. That is, these do not return "UTF-8" as the encoding for any font in either mode.

FP_DialogEncodingName

The possible values and the behavior of FrameMaker Dialog Encoding is the same in both *Compatibility Mode* and *Unicode Mode*. Obtaining the value of `FP_DialogEncodingName` by making the call `F_ApiGetString(0, FV_SessionId, FP_DialogEncodingName)` won't return "UTF-8" even when running FrameMaker on a UTF-8 locale. Therefore, clients that initialize the FDE using the Dialog Encoding must explicitly set the FDE encoding to UTF-8.

Behavior of FDE functions in Unicode Mode

In the FDE, functions found under the following headings in the *FDK Programmer's Reference* do not depend on the FDE encoding and, therefore, have the same behavior in *Unicode Mode* and *Compatibility Mode*:

- Characters
- F-Codes
- Fonts
- Hash Tables
- Memory: manipulating with handles
- Memory: manipulating with pointers
- Metrics
- String lists
- Strings: allocating, copying, and deallocating
- Strings: comparing and parsing
- Strings: concatenating
- Strings: miscellaneous
- Strings: encoded

These include functions that either do not deal with strings (for example `F_MetricSqrt`), expect strings/characters to be in FrameRoman encoding (for example, `F_StrReverse`), can work for any encoding other than UTF-16 or UTF-32 (for example, `F_StrCopyString`), or explicitly ask for the encoding of the strings (for example, `F_StrLenEnc`).

FDE functions found under the following headings in *FDK Programmer's Reference* depend on the FDE encoding and have different behaviors in *Unicode Mode* and *Compatibility Mode*. When *Unicode Mode* is enabled for the FDE, these expect all strings in input parameters to be valid UTF-8 strings. Strings in all return values or parameters set by these functions in this mode will be in UTF-8.

F_FdeInitFontEncs

This call accepts "UTF-8" as an input. Calling this with "UTF-8" as a parameter enables *Unicode Mode* for the FDE. Calling this function with any other encoding disables *Unicode Mode* and enables *Compatibility Mode* for the FDE.

F_Printf

This function accepts the %C escape sequence. In *Compatibility Mode*, this ignores the corresponding parameter. In *Unicode Mode*, the first UTF-8 character in the corresponding parameter (which must be `ConStringT` or `UCharT *`) is printed. The following code prints $\forall + \varrho = 6$

```
...
StringT devanagiri_four = "\xE0\xA5\xAA";
StringT devanagiri_two = "\xE0\xA5\xA8";
IntT res;
F_FdeInitFontEncs((ConStringT)"UTF-8");
res = F_DigitValue(devanagiri_four)
+F_DigitValue(devanagiri_two);
F_Printf(NULL, "%c + %c=%d", devanagiri_four, devanagiri_two,
res);
...
```

F_FontEncName, F_FontEncId

These functions can handle UTF-8. For example, the following code prints UTF-8

```
...
FontEncIdT feId;
F_FdeInitFontEncs((ConStringT) "UTF-8");
feId = F_TextEncToFontEnc(F_EncUTF8);
if (feId == F_FontEncId("UTF-8"))
F_Printf(NULL, F_FontEncName(feId));
...
```

NOTE: These functions behave the same in *Unicode Mode* and *Compatibility Mode*.

Debugging

The following API functions found in the *FDK Programmer's Reference* under the heading *Debugging* are a notable exception to conventions. These behave as a part of the FDE, rather than as a part of the APIs, and thus depend upon the FDE encoding. Their behavior is similar to the behavior of `F_Printf` (see under the I/O section).

- F_ApiPrintTextItem
- F_ApiPrintTextItems
- F_ApiPrintPropVal
- F_ApiPrintPropVals

Files, directories, and filepaths

These functions can deal with Unicode paths when Unicode Mode is enabled. Like everywhere else, all inputs are expected to be in UTF-8 and all outputs are in UTF-8.

I/O

The calls F_ChannelOpen and F_ChannelMakeTmp can accept UTF-8 filenames when in *Unicode Mode*. They handle filepaths in the same manner as FilePath functions discussed above. The calls F_Printf and F_Warning also require UTF-8 input in *Unicode Mode*.

In Unicode Mode, all data written to the NULL channel (the console) using F_Printf, F_Warning, or F_ChannelWrite must be in UTF-8. Because the console window in the Windows platform can display Unicode irrespective of the locale, the UTF-8 input is displayed identically across all locales.

Maker Interchange Format (MIF)

These calls can accept UTF-8 input when in *Unicode Mode*. In *Unicode Mode*, F_MifString writes all FM_Tab, FM_NonBrkHyphen, FM_DiscHyphen, and FM_HardSpace in a string by starting a separate character tag. So a string "space-time" with a nonbreaking hyphen between "space" and "time" will be written as follows by F_MifString:

```
<String `space`>  
<Char HardHyphen>  
<String `time`>
```

String handling functions in FDE

This section explains how string handling functions in FDE work in the *Unicode Mode*.

Handling of Unicode Characters

Many calls such as `F_StrChrUTF8` and `F_CharIsLowerUTF8` accept a Unicode character in UTF-8 encoding. The character can be passed to these functions in the form of a `UCharT` pointer, a `StringT`, or a `ConStringT`. In the following example, the function `F_CharUTF8ToUTF32` treats the sequence `0xE2 0x80 0x93` as a single character EM DASH '—' in UTF-8:

```
#include "fencode.h"

...

UChar32T emDash_UTF32;
UCharT emDash[3];
emDash[0]=0xE2;
emDash[1]=0x80;
emDash[2]=0x93;
emDash_UTF32 = F_CharUTF8ToUTF32((ConStringT) emDash);
```

The functions expecting Unicode characters in this manner only parse the `UCharT` sequence enough to pick up one character. Therefore, the `UCharT` sequence need not be terminated by a null byte when being passed as a character. If a `UCharT` sequence contains more than one UTF-8 character, only the first character is considered. You must provide valid sequences containing at least one Unicode character to these functions.

Truncation of Unicode Strings

All FDE functions, unless stated otherwise, expect and return lengths of UTF-8 strings in terms of bytes, rather than the number of Unicode characters. Some FDE string handling functions either restrict a string to a certain number of bytes, or consider the strings only up to a certain number of bytes for performing comparisons or other operations. An example of this is the `StrCpyN` function, which copies at most `N` bytes (including the terminating null byte) from one string into another.

Some of these functions might truncate a UTF-8 string at an invalid boundary. In the following example, `F_StrTrunc` truncates the UTF-8 string "A—B" at an invalid boundary, rendering it invalid. The effect of this call is to truncate the string to `"\x41\xe2\x80"` (midway between the EM DASH character '—').

```
StringT s = F_StrCopyString((ConStringT)
"\x41\xe2\x80\x93\x42");
F_StrTrunc(s, 3);
```


Such functions aren't safe for UTF-8 input. Certain functions of this type that have been labeled UTF-8 safe do not truncate strings at an invalid boundary. For example, `F_StrTruncEnc` is UTF-8 safe when called with UTF-8 as the encoding.

```
StringT s = F_StrCopyString((ConStringT)
"\x41\xe2\x80\x93\x42");
FontEncIdT feId = F_FontEncId((ConStringT) "UTF-8");
F_StrTruncEnc(s, 3, feId);
```

The above call truncates the string to `"\x41"` (or `"A"`), which is the last complete UTF-8 character in the invalid string `"\x41\xe2\x80"`. In a similar manner, in the following call, `F_StrCmpNEnc` returns `True`, while `F_StrCmpN` doesn't for the same strings `"ÆÐ"` and `"ÆØ"`, and the same length 3.

```
StringT s1 = F_StrCopyString((ConStringT) "\xe2\x80\x93");
StringT s2 = F_StrCopyString((ConStringT) "\xe2\x80\x93");
FontEncIdT feId = F_FontEncId((ConStringT) "UTF-8");
if (F_StrCmpNEnc(s1,s2,3,feId)==0)
F_Printf(NULL,(ConStringT) "\nF_StrCmpNEnc:%s and %s are equal
on %d bytes", s1, s2, 3);
else
F_Printf(NULL,(ConStringT) "\nF_StrCmpNEnc:%s and %s are not
equal
on %d bytes", s1, s2, 3);
if (F_StrCmpN(s1,s2,3)==0)
F_Printf(NULL,(ConStringT) "\nF_StrCmpN:%s and %s are equal on
%d bytes", s1, s2, 3);
else
F_Printf(NULL,(ConStringT) "\nF_StrCmpN:%s and %s are not equal
on %d bytes", s1, s2, 3);
```

The code produces the following output on the console window:

```
F_StrCmpNEnc:ÆÐ and ÆØ are equal on 3 bytes
F_StrCmpN:ÆÐ and ÆØ are not equal on 3 bytes
```

Special remarks on UTF-16 and UTF-32

No FDK API can handle UTF-16 or UTF-32 input. FDE functions that can handle UTF-16 and UTF-32 input expect the input to be in the endianness of the Operating System unless explicitly stated otherwise. Therefore, these functions expect the input to be in UTF-16LE and UTF-32LE.

Because a code unit in UTF-16 and UTF-32 encodings is 2 and 4 bytes respectively, a string in these encodings needs the sequence 0x00 0x00 and 0x00 0x00 0x00 0x00 respectively to indicate string termination. Single 0x00 bytes might occur multiple times in a string without being interpreted as a terminating character. For example, the string "AB" is represented as 0x41 0x00 0x42 0x00 in UTF-16LE. Therefore, many functions like `F_StrLen` that treat a single occurrence of 0x00 as an indication of string termination won't work correctly on UTF-16 and UTF-32 strings. Therefore, a function that has been explicitly indicated as UTF-16 or UTF-32 safe should not be passed strings in these encodings.

UTF-8 string support in FDE functions

The following FDE functions should *not* be used for UTF-8 strings as they can truncate a UTF-8 string midway through a character, rendering it invalid:

```
F_StrCpyN, F_StrCmpN, F_StrEqualN, F_StrICmpN, F_StrIEqualN,
F_StrPrefixN, F_StrCatN, F_StrCatIntN, F_StrCatCharN,
F_StrTrunc
```

The following functions also do not handle Unicode properly because they work on a bitwise basis and are UTF-8 unsafe:

```
F_StrTok, F_StrBrk, F_StrChr, F_StrRChr, F_StrReverse,
F_StrStrip
```

The following functions also work on a byte-wise basis but are UTF-8 safe. They do not render UTF-8 strings invalid and can work for UTF-8 input to some extent. Because of the nature of UTF-8 encoding, `F_StrCmp` and `F_StrEqual` can be used to check Unicode code point-based equality. The functions `F_StrPrefix`, `F_StrSubString`, and `F_StrSuffix` also work correctly if given valid UTF-8 input. The case-insensitive versions of these functions are safe as well but can only take the case of the English alphabet (A-Z) into consideration.

```
F_StrCmp, F_StrEqual, F_StrICmp, F_StrIEqual, F_StrPrefix,
F_StrIPrefix, F_StrSubString, F_StrSuffix
```

The following functions can be used for Unicode without any problems:

```
F_StrStripLeadingSpaces, F_StrStripTrailingSpaces, F_StrNew,
F_StrCopyString, F_StrCpy, F_Free, F_ApiDeallocateString,
F_StrCat, F_StrLen
```

The following functions can also be used for Unicode but won't deal with numbers written in different scripts (like Hindi and Arabic) properly:

```
F_StrAlphaToInt, F_StrAlphaToReal
```

UTF-8 handling by "Enc" functions

FDE string handling functions that have an `Enc` suffix can handle double-byte encodings as well as UTF-8 encoding. The exceptions are:

```
F_StrChrEnc, F_StrRChrEnc, F_StrCatDblCharNEnc
```

The Enc functions that can accept UTF-8 never cut a Unicode string midway of a character's bytes and ensure that the string is valid if the input was valid. So, a function like `F_StrCpyNEnc` that copies at most N-1 bytes might copy fewer if cutting the string at the N-1 character makes it invalid. These functions cut strings at (or consider the string until) the first valid boundary before the point specified. These include:

`F_StrTruncEnc`, `F_StrCatNEnc`, `F_StrNCatNEnc`, `F_StrCpyNEnc` and
`F_StrCmpNEnc`, `F_StrICmpNEnc`, `F_StrIEqualNEnc`

The "Enc" functions compare Unicode characters by the code points (character-by-character, not byte-by-byte). Case-insensitive comparison is done by conversion to lower case followed by comparison on code points. Because of the UTF-8 encoding design, byte-by-byte comparison is equivalent to code-point-by-code-point comparison. The functions that perform comparisons are:

`F_StrIEqualEnc`, `F_StrIEqualNEnc`, `F_StrICmpEnc`, `F_StrMCompEnc`,
`F_StrCmpNEnc`, `F_StrICmpNEnc`, `F_StrQsortCmpEnc`,
`F_StrIPrefixEnc`, `F_StrISuffixEnc`, `F_StrStrEnc`

The function `F_StrLenEnc` returns the number of Unicode characters in the string. Use `F_StrLen` to get the number of bytes.

Compatibility mode

The APIs and FDE functions provide backward compatibility to a large extent in *Compatibility Mode*, but there are certain limitations as mentioned in the following sections.

FDK 8 is slower in *Compatibility Mode* than in Unicode Mode. The slowness and limitations of *Compatibility Mode* make Unicode Mode the recommended mode of operation.

When running in the *Compatibility Mode*, FDK APIs and FDE functions do not accept UTF-8 input and mimic the behavior of FDK 7.2 as closely as possible. APIs and FDE functions that existed in FDK 7.2 exhibit such behavior for any new properties as well. However, new APIs and functions added in FDK 8 and above exhibit no special behavior in *Compatibility Mode*.

.....
IMPORTANT: *Compatibility Mode should be used, as far as possible, only to support clients that have not been modified to handle Unicode data.*
.....

Enable Compatibility Mode in the FDK

Compatibility Mode is the default mode of operation for both the FDE and APIs and doesn't need to be enabled. If Unicode Mode has not been enabled for the FDE or APIs, they are running in *Compatibility Mode*.

This allows clients written for releases compiled with FDK 7.2 or earlier to function correctly with FrameMaker 8 and above without being recompiled. This also minimizes the changes that clients need if they are recompiled with FDK 8 and above.

The APIs and FDE functions provide backward compatibility to a large extent in *Compatibility Mode*, but there are certain limitations as mentioned in the following sections.

You can explicitly set the APIs to work in *Compatibility Mode* if they were running in *Unicode Mode* by making the following call:

```
F_ApiEnableUnicode(False);
```

You can explicitly set the FDE to work in *Compatibility Mode* if it was running in *Unicode Mode* by setting the FDE encoding to any encoding other than "UTF-8" by calling `F_FdeInitFontEncs`. For example, the following code sets the FDE encoding to "FrameRoman", enabling *Compatibility Mode* for the FDE:

```
F_FdeInitFontEncs((ConStringT) "FrameRoman");
```

Another example of setting *Compatibility Mode* for the FDE is as follows:

```
StringT encName = F_ApiGetString(0,
FV_SessionId, FP_DialogEncodingName);
F_FdeInitFontEncs((ConStringT) encName);
```

The above code sets the encoding of the FDE to the Dialog Encoding of FrameMaker, which is dependent on the Operating System locale and has only five possible values that do not include "UTF-8" (for more details, refer to the Dialog Encoding section later in this document and the *FDK Programmer's Reference*).

NOTE: Running FDE functions or APIs in different modes of operations at different times is called *Mixed Mode* operation and is not recommended. For more details on the *Mixed Mode*, see "The same call produces the following alert on an English locale (where Dialog Encoding is FrameRoman). This is because the byte sequence 0x82 0xA0 0x82 0xD4 0x82 0xA2 stands for the string "Ç†Ç‘Ç€" in FrameRoman encoding." on page 519.

Behavior of FDK APIs in Compatibility Mode

In *Compatibility Mode*, FDK APIs expect all strings to be in the Dialog Encoding of FrameMaker. (See the next section APIs that expect strings in the encoding of the font applied for exceptions to this rule).

The Dialog Encoding of FrameMaker is dependent upon the OS locale settings and has five possible values: FrameRoman, JISX0208.ShiftJIS, BIG5, GB2312-80.EUC, and KSC5601-1992. To find the Dialog Encoding of FrameMaker, make the following call:

```
StringT dialogEnc = F_ApiGetString(0, FV_SessionId,  
FP_DialogEncodingName);
```

APIs that expect strings in the encoding of the font applied

The following APIs do not expect/return strings in the Dialog Encoding of FrameMaker. Instead, they expect/return strings in the encoding of the font applied on the text.

- F_ApiAddText
- F_ApiGetText
- F_ApiGetText2
- F_ApiGetTextForRange
- F_ApiGetTextForRange2

Behavior of F_ApiSave in Compatibility Mode

For clients running FDK APIs in *Compatibility Mode*, F_ApiSave saves documents/books in MIF 7.0 if provided FV_SaveFmtInterchange. To make these clients save documents/books in current MIF format, you must recompile and either enable *Unicode Mode* or provide the new parameter value FV_SaveFmtInterchange100 while calling F_ApiSave.

Internal representation of strings in FrameMaker

The internal representation of all strings in FrameMaker is in UTF-8. Hence, the name of a paragraph format, once set in a FrameMaker document (FM/MIF), is displayed in the same manner across all locales.

When APIs are used to obtain strings that aren't representable in the Dialog Encoding of FrameMaker

FrameMaker can have characters from scripts like Arabic, Devanagiri, and others. When queried using an API in *Compatibility Mode*, strings containing such characters always contain question marks '?' instead of the characters because these aren't valid in any of the five Dialog Encodings.

Japanese strings are also stored internally as UTF-8 but have valid representations in Shift-JIS. Therefore, on a Japanese locale, when queried using an API in *Compatibility Mode*, the Shift-JIS equivalent of such strings are returned correctly. However, on an English locale, when queried in the same manner, the strings contain question marks '?' in place of any character that isn't representable in FrameRoman. No Japanese character, for example, is representable in FrameRoman. However, because Shift-JIS also contains the basic ASCII range, which is representable in FrameRoman, characters are converted correctly.

Passing a string not in the Dialog Encoding of FrameMaker

If FDK APIs are running in *Compatibility Mode* on an English locale and you provide the input to an API in Shift-JIS encoding, this is incorrect because the API interprets the input as FrameRoman encoding. Even if the string is valid in the current Dialog Encoding, its interpretation in terms of glyphs is incorrect. For example, the Shift-JIS string " あぶい " has the byte representation 0x82 0xA0 0x82 0xD4 0x82 0xA2, which is a valid byte sequence in FrameRoman encoding, but is actually interpreted as the string "ç†ç`çç" in FrameRoman encoding. When this input is passed to an FDK API in *Compatibility Mode* on an English locale, the API internally converts it to UTF-8 assuming it to be FrameRoman. If the API is used to set the paragraph format name and the file generated is subsequently stored as a FrameMaker document (FM/MIF) and then opened on ANY locale, the paragraph format name displays as "ç†ç`çç" (and not as " あぶい " even on a Japanese locale).

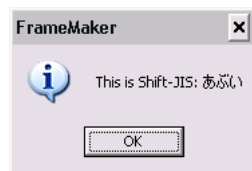
If an API is provided a string in *Compatibility Mode*, a part or whole of which is invalid in the Dialog Encoding of FrameMaker, it is converted to a series of question marks '?'.

Example of an API in Compatibility Mode

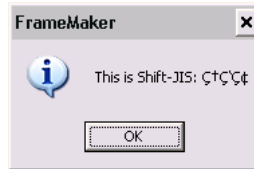
`F_ApiAlert` accepts Shift-JIS input in *Compatibility Mode* if Dialog Encoding is Shift-JIS.

```
F_ApiAlert("This is Shift-JIS: \x82\xa0\x82\xd4\x82\xa2",
           FF_ALERT_CONTINUE_NOTE);
```

This call produces the following alert on a Japanese locale (where Dialog Encoding is Shift-JIS):



The same call produces the following alert on an English locale (where Dialog Encoding is FrameRoman). This is because the byte sequence 0x82 0xA0 0x82 0xD4 0x82 0xA2 stands for the string "Ç†Ç‘Ç€" in FrameRoman encoding.



F_ApiNotify

In *Compatibility Mode*, the string parameter `sparm` is in the Dialog Encoding. For more information on filepath handling in *Compatibility Mode*, see the section on FDE FilePath functions.

F_ApiGetSupportedEncodings

In *Compatibility Mode*, the list returned by the API *F_ApiGetSupportedEncodings* doesn't contain "UTF-8" because this encoding isn't supported in the *Compatibility Mode* in any locale. The behavior of this API in *Compatibility Mode* is the same as in FDK 7.2.

F_ApiIsEncodingSupported

In *Compatibility Mode*, `F_ApiIsEncodingSupported` returns False for "UTF-8". The behavior of this API in *Compatibility Mode* is the same as in FDK 7.2.

F_ApiSave, F_ApiOpen, F_ApiImport

In *Compatibility Mode*, the filename is expected to be in the Dialog Encoding. For more information on filepath handling in *Compatibility Mode*, see the section on FDE FilePath functions.

In the *Compatibility Mode*, `FV_SaveFmtBinary` and `FV_SaveFmtInterchange` are equivalent to `FV_SaveFmtBinary80` and `FV_SaveFmtInterchange70` respectively when calling `F_ApiSave`. Therefore, the default mode of saving MIF in *Compatibility Mode* is legacy format (MIF7 and MIF Book 7). The default mode for saving FM files is FrameMaker 11 format (FM 11 and FM Book 11).

F_ApiAddText F_ApiGetText, F_ApiGetText2, F_ApiGetTextForRange, F_ApiGetTextForRange2

In *Compatibility Mode*, these APIs do not expect/return strings in the Dialog Encoding of FrameMaker. Instead, they expect/return strings in the encoding of the font applied on the text. Therefore if the font MS PMincho (encoding of Shift-JIS) is applied to the text " あぶい ", `F_ApiGetText` returns the byte sequence 0x82 0xA0 0x82 0xD4

0x82 0xA2, which is the representation of "あぶい" in Shift-JIS. The same document can have the text "ÆÐ" with the font Times New Roman (encoding of FrameRoman) applied. For this text, `F_ApiGetText` returns 0xAE 0xC3, which is the FrameRoman encoding of "ÆÐ". Characters that are not representable in the encoding of the applied fonts are converted to question marks. This behavior is common to all variants of `F_ApiGetText`.

`F_ApiAddText` has a similar behavior. It expects the input to be in the encoding of the font applied at the insertion point. Thus, if MS PMincho is applied at the insertion point, `F_ApiAddText` expects the input to be in Shift-JIS encoding. The font must be applied to a location before the addition of text so that `F_ApiAddText` can ascertain the encoding. If all the document fonts are in the same encoding as the text (including the default paragraph font), this will pose no problems. However if a document is likely to have mixed contents, for example if it has some Japanese content "あぶい" in MS PMincho (byte representation 0x82 0xA0 0x82 0xD4 0x82 0xA2) right after some English content in Times New Roman, the font MS PMincho must be applied to the insertion location before adding the text. Otherwise, it is interpreted as being in FrameRoman encoding and is added to the document as the text "あぶい".

The conversion from UTF-8 to single byte and vice-versa that must be performed for Symbol/Dingbats/Webdings in *Unicode Mode* is already handled by the API in *Compatibility Mode* (see the information for these same APIs in the Unicode Mode section). Therefore, getting the text ∇ (the Ñ with Symbol font applied) returns 0xD1, and not 0xC3 0x91, in *Compatibility Mode*. Similarly, for adding this character (when the font is Symbol), `F_ApiAddText` expects 0xD1, and not 0xC3 0x91.

F_ApiGetEncodingForFont, F_ApiGetEncodingForFamily

The APIs `F_ApiGetEncodingForFont` and `F_ApiGetEncodingForFamily` have the same behavior as in FDK 7.2 in both *Compatibility Mode* and *Unicode Mode*. That is, these do not return "UTF-8" as the encoding for any font in either mode.

Behavior of FDE functions in Compatibility Mode

In the FDE, functions found under the following headings in the *FDK Programmer's Reference* do not depend on the FDE encoding and, therefore, have the same behavior in *Unicode Mode* and *Compatibility Mode*:

- Characters
- F-Codes
- Fonts
- Hash Tables
- Memory: manipulating with handles
- Memory: manipulating with pointers

- Metrics
- String lists
- Strings: allocating, copying, and deallocating
- Strings: comparing and parsing
- Strings: concatenating
- Strings: miscellaneous
- Strings: encoded

These include functions that either do not deal with strings (for example `F_MetricSqrt`), expect strings/characters to be in FrameRoman encoding (for example, `F_StrReverse`), can work for any encoding other than UTF-16 or UTF-32 (for example, `F_StrCopyString`), or explicitly ask for the encoding of the strings (for example, `F_StrLenEnc`).

FDE functions found under the following headings in *FDK Programmer's Reference* depend on the FDE encoding and have different behaviors in *Unicode Mode* and *Compatibility Mode*. When *Compatibility Mode* is enabled for the FDE, these expect all strings in input parameters to be valid in the FDE encoding set by `F_FdeInitFontEncs`. Strings in all return values or parameters set by these functions in this mode will be in the FDE encoding.

- Debugging
- Files, directories, and filepaths
- I/O
- Maker Interchange Format (MIF)

Debugging

The following API functions found in the *FDK Programmer's Reference* under the heading *Debugging* are a notable exception to conventions. These behave as a part of the FDE, rather than as a part of the APIs, and thus depend upon the FDE encoding. Their behavior is similar to the behavior of `F_Printf` (see under the I/O section).

- `F_ApiPrintTextItem`
- `F_ApiPrintTextItems`
- `F_ApiPrintPropVal`
- `F_ApiPrintPropVals`

Files, directories, and filepaths

In *Compatibility Mode*, all inputs are expected to be in FDE encoding and all outputs are also in this encoding. The system locale must also be compatible with the FDE encoding. For example, if the FDE encoding is "JISX0208.ShiftJIS", the locale must be Japanese for Japanese filenames to be used correctly.

Filepaths that contain invalid characters in the FDE encoding are inaccessible.

I/O

The calls `F_ChannelOpen` and `F_ChannelMakeTmp` handle filepaths in the same manner as filepath functions discussed above. All data written to the NULL channel using `F_Printf`, `F_Warning` or `F_ChannelWrite` goes to the console. Because the console window in the Windows operating system expects Unicode irrespective of the locale, data sent to the console is converted from the FDE encoding to UTF-8 before being displayed.

Hence, these calls expect the data to be in the FDE encoding when writing to the console.

Maker Interchange Format (MIF)

In *Compatibility Mode*, these functions behave as they did in FDK 7.2. Note that only `F_MifText` is capable of dealing with double-byte strings, and the rest expect `FrameRoman` strings. None of these functions depend on the FDE encoding in *Compatibility Mode*.

Structured Import/Export APIs

The FDK Structured APIs do not provide a *Compatibility Mode* of operation. The behavior of these APIs is dependent on two aspects: the statically linked code that resides in `struct.lib` shipped with the FDK, and the FDK APIs that this code internally calls.

The FDK APIs provide a *Compatibility Mode*, which is the default mode of operation until Unicode Mode is enabled by making the `F_ApiEnableUnicode(True)`. However, the static code in `struct.lib` in FDK doesn't have a *Compatibility Mode*, and these APIs might fail or produce unpredictable results with non-Unicode data.

If you want legacy behavior (*Compatibility Mode*) in structured clients, you must use the `struct.lib` shipped with FDK 7.2 or earlier. In addition, you must run FDK APIs in *Compatibility Mode*. Legacy clients that have not been recompiled with FDK run in *Compatibility Mode* without a problem. For *Unicode Mode*, you must link the client against `struct.lib` shipped with FDK 11 and run FDK APIs in *Unicode Mode*.

International Components for Unicode (ICU)

The FDE functions rely extensively on International Components for Unicode (ICU). This has some important consequences for both legacy and current FDK clients.

Set the ICU data directory

For correct functionality, ICU requires convertor data. Any process that uses ICU must initialize ICU by setting its data directory. FrameMaker also uses ICU internally and initializes it by setting the data directory to `fminit/icu_data` where all ICU convertor data shipped with FrameMaker is stored. Because the FDE uses ICU extensively, all clients (including asynchronous clients) compiled with FDK must also set the ICU data directory correctly by making the following call (where `icu_data_dir` is the directory that stores the ICU convertor data). The ICU data directory must be in ASCII only. A network pathname (UNC) can be used as the data directory.

```
F_SetICUDataDir(icu_data_dir);
```

Setting the ICU data directory has a process-wide effect. For example, because synchronous DLL clients on Windows reside in the same process as FrameMaker, which also initializes ICU for internal usage, such clients do not need to set the ICU data directory explicitly. Setting the ICU data directory incorrectly can adversely affect FrameMaker if the client is in the same process space. Use the call `F_GetICUDataDir` to query the current ICU data directory being used within a process.

Clients that do not reside in the same process space as FrameMaker must set the ICU data directory for correct functionality. Wherever possible, `F_FdeInit` attempts to set the ICU data directory if it has not already been set. Because `F_FdeInit` attempts to pick up the ICU data directory path information from the instance of FrameMaker that the client is connected to, it can set ICU data directory properly only when the client is connected to a FrameMaker session running on (and from) the same machine at the time of the call.

Clients should set the ICU data directory themselves. This is particularly important for remote clients and asynchronous clients that sometimes make FDE calls without being attached to a FrameMaker session.

F_FdeInit as the first FDE call

Using FDK 7.2, you could sometimes write clients that did not call `F_FdeInit` or that called other FDE functions first.

However, from FDK 8 onwards, the dependence on internal FDE structures being initialized is greater, and `F_FdeInit` attempts to initialize the ICU data directory for the client if it has not already been set. Therefore, you must make the `F_FdeInit` call before making any other FDE call.

Exceptions are `F_SetICUDataDir` and `F_GetICUDataDir`, which you can call before calling `F_Fdelnit`.

Dependency on ICU DLL files at run time

All FDK clients need ICU DLLs at run time. For synchronous clients, `FrameMaker` takes care of loading these DLLs. Asynchronous clients must make sure that the required ICU DLLs are present in any of the following locations:

- A directory which is in the system search path
- The directory where the client's executable resides
- The directory from which the client is executed

ICU DLL files are shipped with `FrameMaker` and can be found in the `FMHome` folder. These are also available for download from ICU website.

Mixed Mode operations

Using the calls mentioned in the previous sections, you can run APIs in Unicode Mode while running the FDE in Compatibility Mode, and vice versa. The resulting mode is called a *Mixed Mode*. You can also keep switching between Unicode Mode and Compatibility Mode. This mode is also called a *Mixed Mode* because the client runs in different modes at different points in time.

Such Mixed Mode operations are potentially hazardous because they may result in inconsistently encoded strings. For example, a filepath enumeration code created using the FDE function `F_FilePathGetNext` returns the filenames in UTF-8 if the FDE is running in Unicode Mode. However, if FDK APIs are running in Compatibility Mode, `F_ApiOpen` can't open the filename provided by `F_FilePathGetNext` because it can't handle Unicode in Compatibility Mode.

.....
IMPORTANT: *Mixed Mode operations aren't recommended and can result in unpredictable behavior. FDK API and FDE modes must be changed together and before making calls to any other APIs or FDE functions (with the exception of `F_SetICUDataDir` and `F_GetICUDataDir`).*

Handling for special characters

This section describes how special characters are handled in `FrameMaker`, especially in context of UTF encoding.

Special handling for lower 32 characters

The lower 32 characters of any encoding `0x00–0x1F` are used by FrameMaker as special control characters. These are different from ASCII at places. For example, `FC_EOL` or `0x09` is the hard-return character in FrameMaker, which is different from the usual `0x0A` used for the end-of-line character. The lower 32 characters are used uniformly across all encodings. Therefore, EM SPACE, which has the standard Unicode representation of `0x2003`, is represented by `0x14` even in UTF-8 in the context of FDK APIs. These character mappings can be seen in `fcharmap.h` in the FDK include folder.

As a result, some strings returned from APIs aren't strictly in UTF-8 format. In order to convert entirely to UTF-8, any character below 32 would have to be appropriately mapped to a UTF-8 character (for example `0x14` would be mapped to the Unicode code point `0x2003`). Similarly, the APIs do not accept strings truly in UTF-8 format. The character `0x2003` won't behave as EM SPACE in FrameMaker unless it is first converted to `0x14` before being passed to an API. The deviation from Unicode is only for the lower 32 characters.

Sensitivity of certain calls towards special characters

Certain calls, especially `FilePath` and I/O functions in the `FDE` and `F_ApiOpen`, `F_ApiSave`, and `F_ApiImport`, that deal with filepaths are more sensitive towards the presence of certain special characters (below 32 characters).

For example, APIs like `F_ApiOpen` no longer work if characters such as `'\r'` (CR) and `'\n'` (LF) characters are present in the filename. The client should ensure that such special characters are stripped before APIs and FDE functions are called.

Unicode equivalents of special characters

The header file `fcharmap.h` defines characters like `FC_DAGGER` that are in FrameRoman encoding. Equivalent Unicode characters (in UTF-16 format) are also available in the header file. These have been suffixed with a `_U` to indicate that they are the Unicode equivalents of previously defined characters.

Character	Code Point	Description
<code>FC_UTILITY_U</code>	<code>0x01</code>	used by search and index
<code>FC_DBREAK_U</code>	<code>0x02</code>	discretionary break
<code>FC_NBREAK_U</code>	<code>0x03</code>	suppress this break
<code>FC_DHYPHEN_U</code>	<code>0x04</code>	discretionary hyphen

Character	Code Point	Description
FC_NHYPHEN_U	0x05	suppress this h-point
FC_HYPHEN_U	0x06	temporary hyphen
FC_TAB_U	0x08	
FC_EOL_U	0x09	hard return
FC_EOP_U	0x0A	end of para
FC_EOD_U	0x0B	end of flow
FC_SPACE_NUMBER_U	0x10	number space
FC_SPACE_HARD_U	0x11	hard space
FC_SPACE_THIN_U	0x12	thin == 1/12 em
FC_SPACE_EN_U	0x13	en == 1/2 em
FC_SPACE_EM_U	0x14	em == 1 em
FC_HYPHEN_HARD_U	0x15	unbreakable explicit hyphen
FC_ESC_U	0x1B	sentinel code for cblocks
FC_SCH_U	0x1C	sentinel code for sblocks
FC_SPACE_U	0x20	' ' regular space
FC_QUOTEDBL_U	0x22	""straight double quote
FC_QUOTESINGLE_U	0x27	" "straight single quote
FC_BACKSLASH_U	0x5c	"\ "backslash
FC_GUILLEMOTLEFT_U	0x00AB	guillemotleft
FC_GUILLEMOTRIGHT_U	0x00BB	guillemotright
FC_QUOTELEFT_U	0x2018	curly single-left quote
FC_QUOTERIGHT_U	0x2019	curly single-right quote
FC_QUOTEDBLLEFT_U	0x201C	curly double-left quote
FC_QUOTEDBLRIGHT_U	0x201D	curly double-right quote
FC_GUILSINGLLEFT_U	0x2039	guillemotleft
FC_GUILSINGLRIGHT_U	0x203A	guillemotright
FC_QUOTESINGLBASE_U	0x201A	quotesinglbase
FC_QUOTEDBLBASE_U	0x201E	quotedblbase

Character	Code Point	Description
FC_CENT_U	0x00A2	
FC_POUND_U	0x00A3	
FC_YEN_U	0x00A5	
FC_ENDASH_U	0x2013	
FC_DAGGER_U	0x2020	
FC_DAGGERDBL_U	0x2021	
FC_BULLET_U	0x2022	
FC_EMDASH_U	0x2014	
FC_META_U	0x80	

PART IV



Frame Development Environment (FDE)

Introduction to FDE

.....

.....

This chapter provides an overview of how the FDE works and how to use it to write portable FDK clients. It also provides a simple example: a portable filter that you can run right away.

The FDE helps make your clients portable by providing platform-independent alternatives to platform-specific input/output (I/O), string, and memory schemes. With the FDE, you can run your client on all FrameMaker products with minimal effort.

The FDE also provides libraries of utility functions that are useful for filter development.

How the FDE works

The FDE consists of the following:

- A virtual environment
- Utility libraries

Figure 14-1 shows the components of the FDE and their relationship to a client. All of the platform-specific code is contained within the virtual environment.

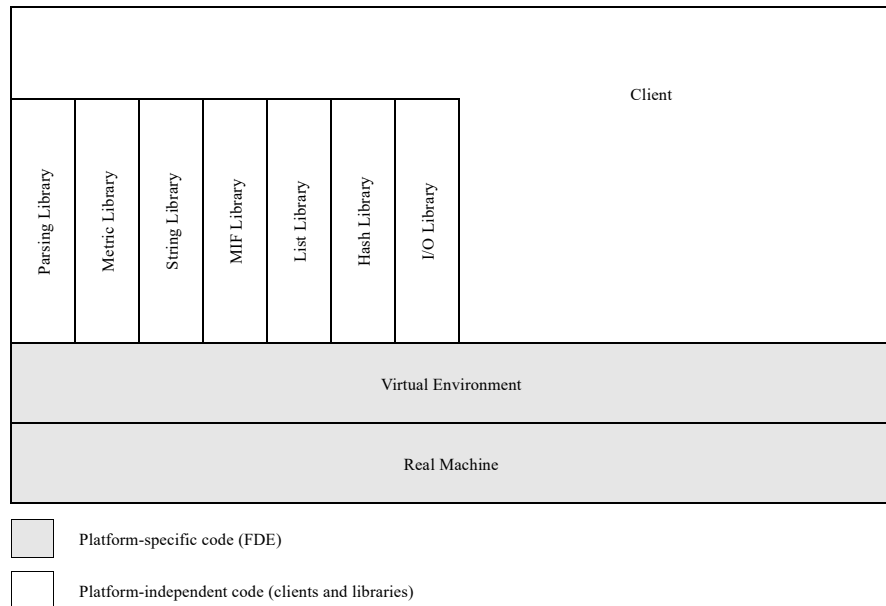


Figure 14-1 *Relationship of the FDE to a client*

The virtual environment

The FDE's virtual environment abstracts the functionality of platform-dependent compilers, operating systems, and C language libraries. It provides the following facilities to replace platform-dependent schemes:

- **I/O channels**
Channels are an abstraction of I/O streams. Instead of directly using the input, output, and temporary files of a particular environment, you use the virtual environment's channels.
- **Memory handling**
The virtual environment provides platform-independent memory allocation and deallocation functions for pointers and handles.

Utility libraries

The FDE utility libraries provide the following types of functions:

- Platform-independent replacements for C language library functions, such as `strcpy()` or `strlen()`
- `MetricT` type functions for converting and manipulating FrameMaker product `MetricT` values

The `MetricT` data type is used in the API to express measurements, such as font sizes and line widths. It is also used in MIF `<MathFullForm>` statements. For more information on the `MetricT` data type, see “MetricT values” in the FDK Programmer’s Reference.

- MIF writing functions for directing output to a MIF output channel, setting indent levels for the channel, and writing a variety of simple MIF statements
- I/O functions for writing and reading from input channels when byte swapping is necessary

How to make your client portable

To use the FDE to make your client portable, follow these general steps:

- 1 *Add a call to `F_FdeInit()` to initialize the FDE environment.*

To initialize the FDE environment, call `F_FdeInit()` as follows:

```
F_FdeInit();
```

- 2 *Replace C primitive data types with FDE types.*

Replace any C primitive data types, such as `char` or `int`, with FDE types, such as `UCharT` or `IntT`.

- 3 *Replace I/O and memory calls in your client with calls to the FDE’s virtual environment.*

Use the FDE virtual environment’s channel I/O and memory functions to make your I/O and memory calls platform independent. For a description of the virtual environment and its I/O and memory facilities, see Chapter 15, “Making I/O and Memory Calls Portable.”

- 4 *Replace string, character, and other platform-specific C library function calls with corresponding calls to FDE utility library functions.*

For information on the utility library functions, see the chapter, “FDK Function Reference,” of the FDK Programmer’s Reference.

5 *Include the appropriate FDE header files.*

All FDE clients must include the `fdetypes.h` header file. If you are using functions from the FDE utility libraries, you must also include the header files for these functions after `fdetypes.h`. For example, you must include `fstrlist.h` if you use any of the string list functions. If you need to include other C library header files, they must precede all FDE header files.

6 *Compile your client and link the FDK library with it.*

The following sections describe Steps 2, 3, and 4 in greater detail.

Replacing C primitive data types with FDE types

To ensure portability across different platforms and compilers, the FDE uses substitutes for C language primitive data types. These types are defined in the `f_types.h` header file. The following table lists the FDE data types and their equivalents.

FDE data type	Equivalent type	Size
AddrT	<code>char*</code> , <code>void*</code>	Unsigned 4 bytes
BoolT	<code>long</code>	Signed 4 bytes
ByteT	<code>char</code>	Signed 1 byte
CharT	<code>char</code>	Signed 1 byte
ConStringT	<code>const unsigned char*</code>	Pointer
ErrorT	<code>long</code>	Signed 4 bytes
FunctionT	Function pointer (returns <code>IntT</code>)	Unsigned 4 bytes
GenericT	<code>char*</code> , <code>void*</code>	Signed 4 bytes
IntT	<code>long</code>	Signed 4 bytes
MetricT	<code>long</code>	Signed 4 bytes
NativeDoubleT	<code>double</code>	Signed 4 bytes (platform dependent)
NativeIntT	<code>int</code>	Signed 4 bytes (platform dependent)
NativeCharT	<code>char</code>	Signed 1 byte (platform dependent)
NativeULongT	<code>unsigned long</code>	Unsigned 4 bytes (platform dependent)
NativeLongT	<code>long</code>	Signed 4 bytes (platform dependent)
ProcedureT	Procedure pointer	Unsigned 4 bytes

FDE data type	Equivalent type	Size
PByteT ^a	int, char	Signed 4 bytes
PCharT	int, char	Signed 4 bytes
PRealT	double	Signed 4 bytes
PShortT	int, short	Signed 4 bytes
PtrT	char*, void*	Unsigned 4 bytes
PUByteT	unsigned int unsigned char	Unsigned 4 bytes
PUCharT	unsigned int unsigned char	Unsigned 4 bytes
PUShortT	unsigned int, unsigned short	Unsigned 4 bytes
RealT	float	Signed 4 bytes
ShortT	short	Signed 2 bytes
StrBuffT	char [STRBUFFSIZE+1]	256 bytes
StringT	unsigned char*	Pointer
UByteT	unsigned char	Unsigned 1 byte
UCharT	unsigned char	Unsigned 1 byte
UChar16T	unsigned char	Unsigned 2 bytes
UChar32T	unsigned char	Unsigned 4 bytes
UIntT	unsigned long	Unsigned 4 bytes
UShortT	unsigned short	Unsigned 2 bytes
VoidT	void	None

a. PByteT, PCharT, PRealT, PShortT, PUByteT, PUCharT, and PUShortT are used to suppress compiler errors if a function is not declared in ANSI format. They are used only for function arguments.

To ensure that your client does not use platform-specific data types or functions, the FDE redefines them. If a client that includes the `fdetypes.h` header file uses a platform-specific type or function, the compiler issues an error message when you attempt to compile it. For example, if your client declares the following variable:

```
char ch;
```

the compiler issues an error message similar to the following:

```
#error ! Non_FDE_token "char" ! ch;
```

To avoid these error messages, you can:

- Use the FDE substitute for the platform-dependent data type or function. For example, use `UCharT` instead of `char`.
- Add the following code above the `#include "fdetypes.h"` statement:

```
#define DONT_REDEFINE
```

This prevents the FDE from redefining any data types or functions.
- Use `#undef` to undefine the specific types or functions that you want to use. For example, add the following line after the `#include "fdetypes.h"` statement:

```
#undef char
```

This allows the FDE to generate errors if your client uses any other platform-specific types.

Replacing I/O and memory calls

The following table lists some commonly used I/O and memory calls and the FDE functions you can replace them with:

Function	FDE substitute
<code>fclose()</code>	<code>F_ChannelClose()</code>
<code>fopen()</code>	<code>F_ChannelOpen()</code>
<code>fwrite()</code>	<code>F_ChannelWrite()</code>
<code>alloc()</code>	<code>F_Alloc()</code>
<code>free()</code>	<code>F_Free()</code>
<code>printf()</code>	<code>F_Printf()</code>
<code>sprintf()</code>	<code>F_Sprintf()</code>

Some FDE functions have slightly different parameters or return values than the corresponding platform-specific I/O and memory functions. For example, `F_Alloc()` has a parameter that `alloc()` doesn't have. Before using an FDE I/O or memory function, look it up in the chapter, "FDK Function Reference," of the FDK Programmer's Reference.

Replacing C library calls

The following table lists some commonly used C library functions and the FDE functions you can replace them with:

Function	FDE substitute
<code>strcmp()</code>	<code>F_StrEqual()</code> or <code>F_StrCmp()</code>
<code>strlen()</code>	<code>F_StrLen()</code>
<code>strcpy()</code>	<code>F_StrCpy()</code>
<code>strcat()</code>	<code>F_StrCat()</code>

Some FDE functions have slightly different parameters or return values than the corresponding C library functions. For example, `F_StrCpy()` returns `VoidT`, while `strcpy()` returns a pointer. Before using an FDE library function, look it up in the chapter, "FDK Function Reference," of the FDK Programmer's Reference.

A simple FDE filter

The following client filters a text file into a Frame document. Following the code is a line-by-line description of how it works.

```

1  #include "fdetypes.h"
2  #include "fapi.h"
3  #include "fchannel.h"
4  #include "fmemory.h"
5  #include "fmetrics.h"
6  #include "futils.h"
7  #include "fioutils.h"
8
9  #define BUFFERSIZE 1025
10 #define in (MetricT) (72*65536)
11
12 VoidT F_ApiNotify(notification, docId, sparm, iparm)
13     IntT notification;
14     F_ObjHandleT docId;
15     StringT sparm;
16     IntT iparm;
17 {
18     FilePathT *path;
19     ChannelT chan;
20     F_TextLocT tl;
21     StringT buf;
22     IntT count;
23
24     F_FdeInit();
25     buf = F_Alloc(BUFFERSIZE, NO_DSE);
26     if (buf == NULL)
27     {
28         F_ApiBailOut();
29         F_ApiReturnValue(0);
30         return;
31     }
32
33     path = F_PathNameToFilePath(sparm, NULL, FDefaultPath);
34     if (path == NULL) return;
35     if((chan = F_ChannelOpen(path,"r")) == NULL) return;
36
37     docId = F_ApiCustomDoc(F_MetricFractMul(in, 17, 2),
38         11*in, 1, F_MetricFractMul(in, 1, 4), in,
39         in, in, in, FF_Custom_SingleSided, False);
40
41     tl.objId = F_ApiGetId(FV_SessionId, docId, FP_MainFlowInDoc);
42     tl.offset = 0;
43

```

```
44     while(!F_ChannelEof(chan))
45     {
46         count = F_ReadBytes(buf, BUFFERSIZE-1, chan);
47         buf[count] = '\0';
48         t1 = F_ApiAddText(docId, &t1, buf);
49     }
50
51     F_ApiReturnValue(docId);
52     F_FilePathFree(path);
53     F_Free(buf);
54     F_ChannelClose(chan);
55     F_ApiBailOut();
56 }
```

Lines 1 to 11

These lines include header files and define constants for the client.

Lines 12 to 56

These lines define the `F_ApiNotify()` callback, which the FrameMaker product calls when the user attempts to open or import a filterable file. You specify which file types are filterable when you register the filter. For more information on registering the filter, see “Compiling and running the example FDE filter” on page 541.

When the user or a client attempts to open or import a filterable file, the FrameMaker product calls the `F_ApiNotify()` callback with `notification` set to `FA_Note_FilterIn` and `filename` set to the name of the file the user is attempting to open or import.

For more information on `F_ApiNotify()` and other Frame API functions, see the *FDK Programmer’s Reference*.

Line 24

`F_FdeInit()` initializes the FDE. You must initialize the FDE before calling any FDE functions.

Lines 25 to 32

The FDE memory allocation function, `F_Alloc()`, allocates a buffer for reading text from the file. The `NO_DSE` flag instructs the function to return `NULL` if it is unable to allocate memory for the buffer.

The `F_ApiReturnValue(0)` call notifies the FrameMaker product that the filter was unable to filter the specified file. After the filter returns, the FrameMaker product displays an alert informing the user that the filter could not filter the file.

Lines 33 to 34

`F_PathNameToFilePath()` converts the platform-specific pathname specified by `filename` to a platform-independent filepath. The `FDefaultPath` flag instructs `F_PathNameToFilePath()` to use the filenaming conventions of the current platform to interpret the platform-specific pathname. Because the pathname specified by `filename` is absolute, `F_PathNameToFilePath()` ignores the second parameter (the anchor).

Lines 35 to 36

`F_ChannelOpen()` opens the file specified by the platform-independent filepath. The `"r"` instructs `F_ChannelOpen()` to open the file for reading only.

Lines 37 to 40

The filter needs to create a new FrameMaker product document for the filterable file's contents. To do this, it calls `F_ApiCustomDoc()`.

The parameters of `F_ApiCustomDoc()` specify the dimensions and layout of the new document. For example, the first parameter specifies the page width, the `MetricT` equivalent of 8.5 inches. To get the `MetricT` equivalent of fractions such as 8.5 inches, you can't use simple multiplication and division. You must use an FDE metric function, such as `F_MetricMul()`. For example, to get the `MetricT` equivalent of 8.5 inches, you can't use the expression `17/2*in`. You must use `F_MetricFractMul(in, 17, 2)`.

The last parameter of the `F_ApiCustomDoc()` call specifies that the new document is not visible. If another FDK client initiates the open operation, it may keep the document invisible after the filter has added the filterable file's contents to it. If the user initiates the open operation, the FrameMaker product automatically makes the document visible when the filter returns.

Note that you could alternately open an existing document to use as a template for the filtered data. In that case, you should still be sure to open it silently using `F_ApiOpen()`, do not use `F_ApiSimpleOpen()`.

Lines 44 to 50

These lines read text from the input channel and add it to the FrameMaker product document.

`F_ReadBytes()` reads text from the input channel into the buffer until it reaches the end of the file. `F_ApiAddText()` adds the text in the buffer to the Frame document, starting at the current insertion point.

Line 51

This line sets the filter's return value to the ID of the document the filter created. This notifies the FrameMaker product that the filter opened the file successfully.

Lines 52 to 56

These lines clean up and free resources used by the filter. The calls to `F_FilePathFree()` and `F_Free()` free resources used by the filepath and the text buffer; `F_ChannelClose()` closes the input channel; and `F_ApiBailOut()` exits the filter.

Compiling and running the example FDE filter

The source code for the example filter and a makefile or project file are provided online with the FDK. To compile the sample filter, use your platform's make or build utility. For the location of example files and instructions on compiling and linking them, see the *FDK Platform Guide* for your platform.

To run the example filter, follow these general steps:

1 Register the filter:

- Assuming you have compiled your client into a DLL named `filter.dll` and copied or moved it to the FrameMaker product `filters` directory, add the following line to the [APIClients] section of your `product.ini` file:

```
KurtWrite=TextImport,kurt,"KURT",Kurt,filters\filter.dll,krt
```

This instructs the FrameMaker product to call the filter when the user attempts to open or import a file with a `.krt` extension.

- 2 *Create a sample Text Only file.*
Give the file a filename with an `.krt` extension.
- 3 *Start the FrameMaker product.*
- 4 *Open the file you created.*
The FrameMaker product calls the filter, which creates a new document and adds the text from the file to it.

.....

.....

This chapter discusses the FDE virtual environment functions you can use to replace the platform-specific I/O, assertion handler, and memory calls in your client.

For lists of the FDE virtual environment functions, see the chapter, “Function Summary,” of the FDK Programmer’s Reference. For the complete description of a function, look it up in the chapter, “FDK Function Reference,” of the FDK Programmer’s Reference.

Initializing the FDE

Before you call any FDE functions, you should initialize the FDE. To initialize the FDE, call `F_FdeInit()` as follows:

```
.....  
F_FdeInit();  
.....
```

.....
Important: *Each time your client bails out and reinitializes, it should reinitialize the FDE by calling `F_FdeInit()`.*
.....

Using platform-independent representations of pathnames

The FDE allows you to specify pathnames with a platform-independent representation called a *filepath*. The FDE uses the data type `FilePathT` to specify a filepath.

Converting pathnames to filepaths

The FDE provides functions that allow you to convert platform-specific pathname strings to filepaths and filepaths back to platform-specific pathname strings. For example, the following code converts the Windows pathname `\tmp\myfile` to a filepath:

```
. . .
FilePathT *path;
path = F_PathNameToFilePath("/tmp/myfile", NULL, FDosPath);
. . .
F_FilePathFree(path1);
. . .
```

.....
IMPORTANT: If you call a function typed `FilePathT*`, you must use `F_FilePathFree()` to free the returned pointer when you are done with it.

The following code converts the filepath created above back to a Windows pathname:

```
. . .
FilePathT *path;
StringT pathname;
. . .
pathname = F_FilePathToPathName(path, FDosPath);
. . .
. . .
```


To make a `F_FilePathToPathName()` and `F_PathNameToFilePath()` call platform-independent, set the second parameter to `FDefaultPath`. This instructs the function to use the pathname conventions of the platform the client is currently running on. For example, the following code converts a filepath to a pathname for the current platform:

```

. . .
FilePathT *path;
StringT pathname;
. . .
pathname = F_FilePathToPathName(path, FDefaultPath);
. . .
    
```

FDE filepath functions, such as `F_PathNameToFilePath()` and `F_FilePathToPathName()`, have arguments that specify a path type. These arguments are typed `PathEnumT`. `PathEnumT` is defined as:

```

typedef enum{
    FDefaultPath /* Platform the client is running on */
    FDosPath     /* Windows */
    FDIPath      /* Device-independent */
} PathEnumT
    
```

The following table provides examples of the path types.

Path type	Example
FDosPath	c:\mydirect\mysubdir\myfile
FDIPath	<r><c>MyDirectory<c>MySubdirectory<c>MyFile

Device-independent pathnames have the following format:

```
<code>name<code>name<code>name . . .
```

where *code* identifies the role of the component in the pathname and *name* is the name of a component in the pathname. The following table lists codes and their meanings.

Code	Meaning
r	Root of the file tree
c	Component
u	Up one level in the file tree

For example, you can express the following pathname:

```
\MyDirectory\MySubdirectory\MyFile
```

as the following device-independent, absolute pathname:

```
<r><c>MyDirectory<c>MySubdirectory<c>MyFile
```

or as the following device-independent, relative pathname:

```
\<c>MyFile
```

For more information on how a FrameMaker product specifies device-independent pathnames, see the online *MIF Reference*.

Manipulating filepaths

The FDE provides functions that allow you to manipulate filepaths.

For example, `F_DeleteFile()` deletes a file or directory,

`F_FilePathProperty()` checks file permissions, and `F_FilePathGetNext()` allows you to traverse all the files in a

directory. These functions are platform-independent substitutes for functions such as `remove()`, `_access()`, and `_fstat()`.

Making I/O portable with channels

To make your client's I/O portable, you use *channels*. Channels are an abstraction of platform-specific files or file systems. The FDE provides functions to manipulate channels. For example, `F_ChannelOpen()` opens a channel, `F_ChannelRead()` reads from a channel, `F_ChannelWrite()` writes to a channel, and `F_ChannelClose()` closes a channel. These functions are substitutes for platform-specific functions, such as `fopen()`, `fread()`, `fwrite()`, and `fclose()`. All I/O channels in the FDE are buffered internally.

To manipulate a file with FDE channel functions, you must first convert the file's pathname to a filepath and then open the filepath with `F_ChannelOpen()`. For example, the following code opens the file `\myfile` for reading.

```
. . .
ChannelT chan;
FilePathT *path;

path = F_PathNameToFilePath((StringT)"\\myfile",
                             NULL, F_DosPath);
if((chan = F_ChannelOpen(path, "r")) == NULL)
{
    F_Printf(NULL, "Couldn't open file.\n");
    return;
}
. . .
```

Assertion-handler functions

Your client can register an assertion handler and perform its own error handling with the FDE function `F_SetAssert()`. When the client's assertion handler returns, the FDE's assertion handler is called to clean up the system and exit the client properly.

To use the FDE assertion-handler functions, you must include `fdetypes.h` and `fassert.h` in your client.

Making memory allocation portable

The FDE provides a set of functions you can substitute for your client's platform-specific memory allocation and deallocation function calls.

Sometimes you may need to use a pointer directly into absolute memory. To create this pointer, you must first lock the memory to tell the operating system that it should not relocate it. You can then safely use any absolute pointer into the block without fear of the memory being relocated. After you are done with the pointer, you unlock the memory, allowing the operating system to relocate it. To develop portable clients that use large memory blocks, you should use handle-based memory management. There are some trade-offs between using pointers and handles. Handles may slow down the access to memory. Pointers may fragment the heap space. In general, you should use pointers for small memory allocations and handles for large allocations.

Many FDE memory allocation functions, such as `F_Alloc()`, provide a `flags` argument that specifies what to do if memory can't be allocated. If you set this argument to `DSE` and memory can't be allocated, the FDE calls a function that you register by calling `F_SetDSEExit()`. If you set `flags` to `NO_DSE` and memory allocation is unsuccessful, the memory allocation function returns `NULL`. For more information, see "`F_SetDSEExit()`" in the *FDK Programmer's Reference*.

Allocating memory with handles

The FDE provides functions that allow you to allocate and deallocate memory with handles. For example, `F_AllocHandle()` allocates a handle, `F_ClearHandle()` initializes a handle's block of data, and `F_FreeHandle()` frees memory allocated to a handle.

Handle memory functions that are typed `ErrorT` return `FdeSuccess` if they are successful. Other handle memory functions return `NULL` if they are unable to comply with a request.

To use the FDE memory functions, you must include `fdetypes.h` and `fmemory.h` in your program.

Allocating memory with pointers

The FDE provides functions that allow you to allocate and deallocate memory with pointers. For example, `F_Alloc()` allocates memory, `F_ClearPtr()` initializes a pointer's block of data, and `F_Free()` frees memory allocated to a pointer.

Pointer memory functions that are not typed `ErrorT` return `NULL` if they are unable to comply with a request. Functions that are typed `ErrorT` return `FdeSuccess` if they are successful.

To use the FDE memory functions, you must include `fdetypes.h` and `fmemory.h` in your program.

Error and progress reporting

The virtual environment provides functions that allow you to report error and progress status. For example, `F_Warning()` prints a warning message to the Frame console on Windows.

To use the FDE progress reporting functions, you must include `fdetypes.h` and `fprogs.h` in your program.

FDE Utility Libraries

.....

.....

This chapter describes FDE utility libraries:

- The string library provides platform-independent equivalents for many of the functions in `<string.h>`.
- The string list library provides routines for creating a list of strings and manipulating the strings.
- The character library provides platform-independent equivalents for some of the functions in `<ctype.h>` that can be used on Frame characters.
- The I/O library provides functions to read data from or write data to channels and performs byte swapping when it is necessary
- The hash library provides functions to create a hash table and manipulate the cells in it.
- The metric library provides functions for manipulating and converting `MetricT` values (values using Frame’s internal representation of measurements).
- The MIF data structures and macros provide data structures and macros to represent and manipulate the statements described in the online *MIF Reference*.
- The MIF library provides functions that help you write nicely formatted MIF to a channel.
- The simple MIF library provides functions to write individual MIF statements to a channel.

For lists of FDE utility library functions, see the chapter, “Function Summary,” of the FDK Programmer’s Reference. For the complete description of a function, look it up in the chapter, “FDK Function Reference,” of the FDK Programmer’s Reference.

String library

The string library provides functions for allocating and manipulating strings. For example, `F_StrNew()` allocates a string, `F_StrCmp()` compares two strings, and `F_StrAlphaToInt()` converts an alphanumeric string to an integer.

The FDE uses the `StringT` type for strings. `StringT` is an array of `UCharT`. When you specify a string size in an FDE string function, you must include the terminating `0` in the size. The string library provides functions for allocating, manipulating, and freeing strings.

.....
IMPORTANT: Use `F_Free()` to free `StringT` strings. For more information on `F_Free()`, see “*F_Free()*” in the *FDK Programmer’s Reference guide*.

Most FDE string functions have the prefix `F_Str`. To use FDE string functions, you must include `fdetypes.h` and `fstrings.h` in your program and call `F_FdeInit()`.

The string list library

The string list library provides routines for creating a string list and manipulating the strings in it. For example, `F_StrListNew()` allocates a string list, `F_StrListInsert()` inserts a string into a list, and `F_StrListSort()` sorts a string list.

String lists are typed `StringListT`. All the functions in the string list library are prefixed with `F_StrList`. To use the FDE string list functions, you must include `fdetypes.h` and `fstrlist.h` in your program.

Character library

The character library provides routines that convert and manipulate Frame characters. For example, `F_CharIsAlphabetic()` determines whether a character is alphabetic, `F_CharToLower()` converts a character to lowercase, and `F_CharToUpper()` converts a character to uppercase. The FDE implements all character library functions as macros.

The character library functions are all prefixed with `F_Char`. To use the FDE character functions, you must include `fdetypes.h` and `fcharmap.h` in your client.

.....
IMPORTANT: Before you use the character library functions, you must call `F_FdeInit()` to initialize the character library.

The I/O library

The I/O library provides functions that allow your client to read data from and write data to channels. These functions can swap bytes when reading or writing on a channel. For example, it may be necessary to swap bytes in either of the following cases:

- The platform your client is running on is little-endian and the channel you are reading from, or writing to, isn't.
- The channel is little-endian but the platform isn't.

When you use the I/O library functions, you can specify whether an input or output channel's byte ordering is little-endian or big-endian by calling `F_SetByteOrder()` or `F_ResetByteOrder()`. The I/O functions use this information to determine whether byte swapping is necessary. By default, the FDE assumes the channel and the platform are consistent. If you do not specify whether the input or output channel order is little-endian, the functions do not swap bytes.

When you use the FDE I/O functions, you should take care of alignment issues yourself. FDE I/O functions return 0 if they reach the end of a file or an error occurs.

To use the FDE channel utility functions, you must include `fdetypes.h` and `fioutils.h` in your program.

The hash library

The hash library provides functions to create a hash table and manipulate the cells in it. For example, `F_HashCreate()` creates a hash table, `F_HashSet()` adds an entry to a hash table, and `F_HashGet()` fetches an entry from a hash table. All the functions in the library are prefixed by `F_Hash`. To use any FDE hash functions, you must include `fdetypes.h` and `fhash.h` in your program.

The hash table is stored in a data structure of type `HashTableT`. This data structure is opaque to you; you must use the `F_Hash` routines to manipulate a hash table.

Creating hash tables

You create a hash table via `F_HashCreate()`, which is defined as follows:

```
HashT F_HashCreate(StringT name, /* Name of the table */
    IntT minSize, /* Minimum size of the table */
    PShortT keyLen, /* Size of keys */
    GenericT notFound, /* Returned if searched key not found */
/* Determine if cell can be reused */
    BoolT (*deadQuery)(GenericT),
/* Called when cell is deleted */
    VoidT (*removeNotify)(GenericT),
/* Converts key to string*/
    Void(*T stringifyMe)(PtrT, UCharT *));
```

- Use `minSize` to suggest the amount of space the FDE should allocate for the hash table. You can pass a value of 0 which notifies the FDE to use its own calculations for memory allocation.
- For non-string keys, you specify a value for the size of key. For keys that are strings, you specify `KEY_IS_STRING`; in that case each key is a nul-terminated string.
- `F_HashGet()` is a routine that searches for a key and returns the associated data. If it can't find the specified key, it still must return a value; the `notFound` argument specifies what value to return when `F_HashGet()` doesn't find the key.
- `deadQuery` specifies a callback to give you the opportunity to determine the validity of a cell. You should only specify a function for this argument if your code can make this determination; if the function returns `True`, the cell will be marked for deletion. As the FDE hash routines maintain the table, they will call this function to determine whether they can delete the cell's contents. Normally, it is best to specify 0 for this argument, letting the FDE manage the hash table on its own.
- `removeNotify` specifies a callback to invoke whenever an `F_Hash` function or the FDE removes a cell. A typical use for this is to deallocate structures that were allocated for the cell's data. If you don't need to clean up memory for any cells, pass 0.
- `stringifyMe` specifies a procedure that turns non-string keys into strings. This is most useful for debugging.

Structures and pointers in keys

The routines `F_HashSet()`, `F_HashGet()`, and `F_HashRemove()` each receive a key as an argument. These routines work best with flat keys such as integers or strings. It is safest to use flat keys instead of structures for keys; if necessary you can always use `F_Sprintf()` to put structure fields into a string.

The internal hash routine used by these functions examines the bytes in the key. If the key is a structure containing pointers, it only checks the pointers, not the pointed-to data. Further, it examines every byte in the key. If you use structures for keys, you should be sure to clear the memory first, then assign values to the structure members, then set the item in your hash table. For example, use `F_ClearPtr(&myStructKey, sizeof(myStructKey));` when `myStructKey` will be a key for a table cell.

When you pass a key to `F_HashSet()`, the function actually creates a copy of the key. However, it only copies the string, or the number of bytes specified in `F_HashCreate()` as the key length. If your key is a structure that contains pointers, `F_HashSet()` will create copies of the pointers, but not copies of the pointed-to data.

Metric library

The Frame API uses the `MetricT` type to specify measurements, such as tab offsets and font sizes. MIF also uses it in `<MathFullForm>` statements. `MetricT` values should not be confused with the metric system. For more information on metric values, see “MetricT values” in the FDK Programmer’s Reference.

The metric library provides a set of operations that allow you to manipulate metric values without converting them to other units of measurement. For example, `F_MetricFloat()` converts a real number to a metric number, `F_MetricFractMul()` multiplies a metric number by a fraction, and `F_MetricToFloat()` converts a metric number to a real number.

All the functions in the metric library are prefixed with `F_Metric`. To use the FDE metric functions, you must include `fdetypes.h` and `fmetrics.h` in your program.

MIF data structures and macros

The FDE provides data structures that represent the statements described in the online manual *MIF Reference*. It also provides convenience macros that help you get and set fields in these structures. This saves you the effort of developing your own data structures and macros.

MIF data structures adhere to the following naming conventions:

- Structure names contain the prefix `Mif`, the statement name, and the suffix `Struct`. For example, the FDE represents the `<TextFlow>` statement with the `MifTextFlowStruct` structure.
- Structure type names use the suffix `StructT`. For example, the structure type name for `MifTextFlowStruct` is `MifTextFlowStructT`.

- Each of the substatements in a MIF statement is represented as a field, whose name corresponds to the name of the substatement. For example, the FDE represents the statement `<Marker <MType N> <MText string>>` with the structure:

```
typedef struct MifMarkerStruct {
    IntT MType;
    StringT MText;
} MifMarkerStructT;
```

- If a substatement is a structure, a pointer in the structure points to the structure that represents the substatement. For example, the FDE represents the statement `<Para <Pgf>...>` with:

```
typedef struct MifParaStruct {
    MifPgfStructT *Pgf;
    . . .
} MifParaStructT;
```

- If a substatement is a list of structures, it is represented as a link structure. The link structure's name contains a `Mif` prefix, the substatement name, and a `StructL` suffix. The link structure type includes the suffix `StructLT`. A pointer in the structure points to the link structure of the substatement. For example, the FDE represents the statement `<AFrames <Frame>...>` with:

```
typedef struct MifAFrameStruct {
    struct MifFrameStructL *Frame;
} MifAFrameStructT;
```

where the field `Frame` contains a list of all the frames associated with `AFrame`.

To get or set a field in a MIF data structure, use MIF macros. MIF macro names contain the `Mif` prefix, the structure name, the access type (`Get` or `Set`), and the structure field name. For example, to get the `MType` field of a `<Marker...>` statement, use the macro named `MifMarkerGetMType(mif_marker_ptr)`. When you call MIF macros, you must specify pointers to MIF data structures.

The names of MIF macros for list structures comprise the `Mif` prefix, the substatement name, the string `List`, the access type (`Get` or `Set`), and the structure field name. For example, to get `<Polygon... <Point>...>`, you use `MifPointListGetNext`, `MifPointListGetPrev`, and `MifPointListGetPoint`.

Because most MIF data structures contain pointers to other structures, you should dynamically allocate memory space for all data structures to minimize errors.

The MIF library

The MIF library functions help write formatted MIF statements to a channel. For information on functions that write individual MIF statements, see “Simple MIF library” on page 558.

The MIF library maintains its own output channel and indent information. Before you call any MIF library function, you must:

- 1 Call `F_ChannelOpen()` to open a channel.
- 2 Call `F_MifSetOutputChannel()` to set the channel as the MIF output channel.
- 3 Call `F_MifSetIndent()` to set the channel’s indent level.

After you finish writing to the channel, be sure to call `F_ChannelClose()` to close it.

For example, to create a MIF file, use code similar to the following:

```
. . .
#include "fdetypes.h"
#include "fapi.h"
#include "fchannel.h"
#include "futils.h"
#include "fioutils.h"
#include "fmifstmt.h"
. . .
FilePathT *path;
ChannelT chan;

path = F_PathNameToFilePath((StringT)"my.mif",
                             NULL, FDefaultPath);
if((chan = F_ChannelOpen(path,"w")) == NULL) return;
F_MifSetOutputChannel(chan);
F_MifSetIndent(0);
F_MifMIFFile(5.0);
F_MifComment((StringT) "Generated by KurtWrite");
F_MifNewLine();
F_ChannelClose(chan);
. . .
```

This code creates a MIF file named `my.mif` in the default directory (the FrameMaker product directory) with the following contents:

```
<MIFFile 5.00 > # Generated by KurtWrite
```

Some MIF library functions use the type `MifUnitT`, which is an enumerated type specifying measurement units. It can have the following values.

MifUnitT value	Measurement unit
<code>MIFUnitIn</code>	Inches
<code>MIFUnitCm</code>	Centimeters
<code>MIFUnitMm</code>	Millimeters
<code>MIFUnitPica</code>	Picas
<code>MIFUnitPt</code>	Points
<code>MIFUnitDd</code>	Didots
<code>MIFUnitCc</code>	Ciceros
<code>MIFUnitDef</code>	Default unit

The FDE uses `MifUnitT` to generate MIF statements that include decimal values. If `MIFUnitDef` is specified, no unit symbol is generated.

All the MIF library functions are prefixed with `F_Mif`. To use the FDE MIF functions, you must include `fdetypes.h` and `fmifstmt.h` in your program.

Simple MIF library

The simple MIF library functions are useful for writing individual MIF statements. Each function's name corresponds to the MIF statement that it writes.

If a MIF statement uses an enumerated value string, such as `Left`, `Right`, or `Center`, you construct a constant from that string by prefixing it with `MIF`. For example, the enumerated value strings for the `<DParity>` MIF statement are `FirstLeft` and `FirstRight`. When you call the corresponding MIF library function, `F_MifDParity()`, you can specify either `MIFFirstLeft` or `MIFFirstRight` for the `DParity` argument. If you aren't sure if a certain constant exists, you can check for it in the `fmiftype.h` header file.

For the syntax and description of MIF statements, see the online *MIF Reference*.

All simple MIF library functions have the prefix `F_Mif`. To use them, you must include `fdetypes.h` and `fmifstmt.h` in your program.

Glossary

.....

•
•
•
•

This glossary contains words used in the *FDK Programmer's Guide*. For other references providing more information about a term, see the index.

- anchored frame** A container that is tied to a specific location in the text. An anchored frame moves with the text as the anchor symbol (⌵) moves. *See also* unanchored frame.
- bail out** The process by which an API application can exit and free system resources.
- body page** A printable page in a document. *See also* master page.
- building blocks** Text strings that define a cross-reference format, variable definition, index entry, or other item. For example, in a running header (Running H/F) variable, the building block `<$curpagenum>` is a building block that specifies the current page number.
- callback** An application-defined function that responds to a call, such as an initialization call, from a FrameMaker product.
- channel** A platform-independent abstraction of an input or output stream.
- child graphic object** A graphic object that belongs to a particular group or frame.
- draw order** The order in which a FrameMaker product draws graphic objects (that is, the back-to-front order). By default, the draw order is the same as the order in which you draw the graphic objects. When graphic objects overlap, the ones in the front (at the end of the draw order) obscure those in back.
- f-codes** Hexadecimal function codes that specify individual user actions, such as cursor movement or text entry.
- flow** An abstract notion of where text in a document can flow. Flows connect text frames. If text in an autoconnected frame reaches the end of a text frame, the FrameMaker product automatically creates a new text frame and connects it. In Structured FrameMaker documents, a structured flow contains structural elements.
- footnote reference** An anchor that appears in the main text as a number, letter, or special character.
- getting properties** Using API functions to determine and get one or more characteristics of an object.

global document information	Characteristics that apply to an entire document.
graphic object	Anything that the user can create with the Tools palette. Specifically, a graphic object can be an anchored frame, an unanchored frame, a geometric shape (a line, arc, rectangle, rounded rectangle, oval, polyline, or polygon), a group of other graphic objects in a frame together, a text line, a text frame, an imported graphic, an equation, or an inset.
hidden page	The page where a FrameMaker product stores hidden conditional text. A document can have only one hidden page. The API represents each hidden page with an <code>FO_HiddenPage</code> object.
ID	Identifier for an API object. IDs are typed as <code>F_ObjHandleT</code> (32-bit integers).
inset	An imported image created with an inset editor (a specially modified external application that can be launched from a FrameMaker product).
MIF	Maker Interchange Format, a set of ASCII statements that describes a FrameMaker product document or book.
named graphic frames	Frames on reference pages that contain a graphic decoration, such as a line ruling for paragraphs in the body pages or a graphic that appears at the top of the page. The Paragraph Designer provides two settings, Frame Above and Frame Below, to specify the named frames that appear above or below a paragraph.
named objects	API objects identified by a unique name, for example, <code>FO_PgfFmt</code> and <code>FO_CharFmt</code> objects.
object	The API representation of basic entities in a FrameMaker product. For example, an <code>FO_Pgf</code> object represents a paragraph and an <code>FO_Rectangle</code> object represents a rectangle.
page frame	An invisible unanchored frame whose dimensions match those of a page. The API represents a page frame with an <code>FO_UnanchoredFrame</code> object.
parent frame	The frame containing a graphic object. <i>See also</i> page frame.
properties	The characteristics of API objects. Different types of objects have different properties. For example, <code>FO_Rectangle</code> objects have properties to represent a rectangle's height and width. These properties are named <code>FP_Height</code> and <code>FP_Width</code> respectively.
property value	Each property has a value associated with it. For example, if a paragraph has two tabs, the value of its <code>FP_NumTabs</code> property is 2.

reference page	A nonprinting page containing named frames that can be used above and below paragraphs, or above footnote text. A reference page can also contain special flows that control the appearance of generated files such as indexes and lists. <i>See also</i> master page, body page, and named frames.
series object	Objects that the API keeps in ordered series. Series objects include <code>FO_Pgf</code> , <code>FO_BodyPage</code> , and <code>FO_BookComponent</code> objects.
session	Each time you start a FrameMaker product you are starting a session.
type-in properties	Font characteristics and conditional text properties that apply to new text typed at the insertion point.
unique persistent identifier (UID)	An integer that uniquely identifies an object within a document. An object's UID remains the same from one FrameMaker product session to the next.
virtual environment	FDE functionality that abstracts the functionality of platform-dependent compilers, operating systems, and C libraries.

