

Adobe® FrameMaker® 7.2

Structure Import/Export API Programmer's Guide

© Copyright 2005 Adobe Systems Incorporated. All rights reserved.

Adobe® FrameMaker® 7.2 Structure Import/Export API Programmer's Guide

NOTICE: All information contained herein is the property of Adobe Systems Incorporated. No part of this publication (whether in hardcopy or electronic form) may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Adobe Systems Incorporated. The software described in this document is furnished under license and may only be used or copied in accordance with the terms of such license.

This publication and the information herein is furnished AS IS, 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, makes no warranty of any kind (express, implied, or statutory) with respect to this publication, and expressly disclaims any and all warranties of merchantability, fitness for particular purposes, and noninfringement of third party rights.

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, Frame, FrameMaker, PostScript, Acrobat, and Distiller are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Apple, Mac, and Macintosh are trademarks of Apple Computer, Inc., registered in the United States and other countries. Microsoft, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and other countries. JavaScript and all Java-related marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. UNIX is a registered trademark of The Open Group. All other trademarks are the property of their respective owners.

All other trademarks are the property of their respective owners.

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.

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

Table of Contents

Chapter 1 *Introduction*

Who should read this guide?	9	When to develop a structure import/export client	11
What is a structure import/export client?	10	Examples of tasks requiring clients	12
Overview of the import and export process....	10	Understanding the import/export process	13
FmTranslator: A minimal import/export client.	11	Processing events based on their type	16

Chapter 2 *Creating an Import/Export Client*

Understanding FmTranslator	19	Event handlers	23
FmTranslator import and export modules	19	Entity handlers	25
Modifying FDK callbacks	20	Compiling, building, and registering a client.....	28
Creating a custom import/export client	21	Defining a structure import/export client for end	
Header files	22	users	28
Global declarations and constants	22		

Chapter 3 *A Sample Import/Export Client*

Overview of the sample XHTML client	29	Lines 1-21	47
XHTML import	30	Lines 25-133	47
XHTML export example	43	Lines 132-133	48
		Conclusion	48

Chapter 4 *Structure Import/Export API Function Reference*

EndTagOmissible()	50	Structured_GetAppinfo()	57
Structured_CopyAttrVal()	51	Structured_GetDefaultEntityDef()	58
Structured_CopyAttrVals()	53	Structured_GetDelimiterString()	59
Structured_DeallocateAttrVal()	55	Structured_GetDocTypeName()	61
Structured_DeallocateAttrVals()	56	Structured_GetElementDef()	62

Structured_GetEntityDef()	63	Sr_GetAttrVal()	111
Structured_GetEntityNamecase().....	65	Sr_GetAttrVals()	114
Structured_GetFirstElementName()	67	Sr_GetBookCompFilePath().....	116
Structured_GetFirstEntityName()	68	Sr_GetBookFilePath()	117
Structured_GetFirstNotationName().....	70	Sr_GetBookId()	119
Structured_GetGeneralNamecase().....	71	Sr_GetCharFmt().....	120
Structured_GetLcnmchar()	73	Sr_GetChildConvObj()	122
Structured_GetLcnmstrt()	74	Sr_GetColSpecs()	124
Structured_GetLoc()	75	Sr_GetCurColSpecByColNum()	126
Structured_GetNextElementName().....	77	Sr_GetCurColSpecByName()	127
Structured_GetNextEntityName().....	78	Sr_GetCurConvObj()	128
Structured_GetNextNotationName()	80	Sr_GetCurConvObjOfType()	129
Structured_GetNotationDef().....	82	Sr_GetCurSpanSpecByName()	131
Structured_GetOptionalFeature().....	83	Sr_GetDocId()	132
Structured_GetQuantity()	85	Sr_GetExtEntityFilePath()	134
Structured_GetReservedName().....	87	Sr_GetFlowId()	135
Structured_GetUcnmchar()	89	Sr_GetFmChar().....	137
Structured_GetUcnmstrt()	90	Sr_GetFmElemId()	138
Structured_IsAttrCDATA()	91	Sr_GetFmElemTag()	140
Structured_IsAttrFixed()	93	Sr_GetFmObjId()	141
Structured_IsAttrNameToken().....	94	Sr_GetFmText()	143
Structured_IsAttrValSpecified()	95	Sr_GetImportFileHint()	145
Structured_IsIdAttr()	96	Sr_GetInsertedTablePartElementName()	146
Structured_IsIdRefAttr().....	97	Sr_GetInsertLoc().....	147
Sr_AddTableRows()	99	Sr_GetLineBrkInfo()	150
Sr_CancelCurBatchFile().....	100	Sr_GetObjType()	152
Sr_CancelOperation().....	102	Sr_GetParentConvObj()	153
Sr_CellInUse()	103	Sr_GetPrevStructuredGi()	154
Sr_Convert()	104	Sr_GetPrivateData()	155
Sr_DeallocateSessionProps()	105	Sr_GetProcessingFlags()	158
Sr_EventHandler().....	107	Sr_GetPropVal()	159
Sr_GetAssociatedEvent()	109	Sr_GetPropVals()	162

Sr_GetRefElemTag().....	165	Sr_SetRefElemTag().....	221
Sr_GetSessionProps().....	167	Sr_SetSessionProps().....	222
Sr_GetSpanSpecs()	169	Sr_SetSpanSpecs()	224
Sr_GetStraddles().....	170	Sr_SetStraddles()	225
Sr_GetTableCellStartsNewRow().....	172	Sr_SetTableCellUsed()	226
Sr_GetTemplateDocId()	173	Sr_SetTableRowUsed()	227
Sr_GetTextInsetFilePath().....	174	Sr_SetTextInsetFilePath().....	228
Sr_GetTextInsetFlowTag()	176	Sr_SetTextInsetFlowTag()	231
Sr_GetTextInsetFormatting().....	177	Sr_SetTextInsetFormatting().....	232
Sr_GetTextInsetPageSpace()	179	Sr_SetTextInsetPageSpace()	234
Sr_GetVariableName()	180	Sr_SetVariableName()	235
Sr_RowInUse()	182	Sr_UseFmElemId()	236
Sr_SetAttrVal()	184	Srw_CopyColSpec().....	239
Sr_SetAttrVals().....	186	Srw_CopyColSpecs().....	240
Sr_SetBookCompFilePath()	189	Srw_CopyPropVal()	241
Sr_SetBookFilePath().....	190	Srw_CopyPropVals().....	243
Sr_SetBookId()	192	Srw_CopySpanSpec().....	245
Sr_SetCharFmt()	194	Srw_CopySpanSpecs().....	246
Sr_SetColSpecs().....	195	Srw_CopyStraddle().....	247
Sr_SetDocId().....	197	Srw_CopyStraddles()	248
Sr_SetFlowId().....	199	Srw_DeallocateColSpec()	250
Sr_SetFmChar()	202	Srw_DeallocateColSpecs()	251
Sr_SetFmElemTag().....	203	Srw_DeallocatePropVal().....	252
Sr_SetFmText()	205	Srw_DeallocatePropVals()	253
Sr_SetImportFileHint().....	206	Srw_DeallocateSpanSpec()	254
Sr_SetInsertedTablePartElementName().....	209	Srw_DeallocateSpanSpecs()	255
Sr_SetInsertLoc()	210	Srw_DeallocateStraddle()	256
Sr_SetLineBrkInfo()	213	Srw_DeallocateStraddles()	257
Sr_SetPrivateData()	215	Srw_DeleteStraddlesByName()	258
Sr_SetProcessingFlags().....	216	Srw_EntityHandler()	259
Sr_SetPropVal()	217	Srw_GetColSpecByColNum()	261
Sr_SetPropVals().....	219	Srw_GetColSpecByName()	262

Srw_GetExportDtdFilePath()	264	Sw_GetExportFilePath()	309
Srw_GetExportSchemaFilePath()	265	Sw_GetGfxDataAttrVals()	310
Srw_GetImportTemplateFilePath()	266	Sw_GetGfxEntityName()	311
Srw_GetMainDocBookId()	267	Sw_GetGfxEntityType()	312
Srw_GetRulesDocFilePath()	268	Sw_GetGfxNotation()	314
Srw_GetStructuredDeclarationFilePath()	270	Sw_GetGfxPubId()	315
Srw_GetStructuredDocFilePath()	271	Sw_GetGfxSysId()	316
Srw_GetSpanSpecByName()	272	Sw_GetObjType()	317
Srw_LogMessage()	273	Sw_GetParentConvObj()	318
Srw_SetColSpec()	275	Sw_GetPI()	319
Srw_SetSpanSpec()	276	Sw_GetPrivateData()	320
Srw_SetStraddle()	277	Sw_GetProcessingFlags()	323
StartTagOmissible()	278	Sw_GetPropVal()	324
Sw_CancelCurBatchFile()	281	Sw_GetPropVals()	326
Sw_CancelOperation()	282	Sw_GetSessionProps()	329
Sw_Convert()	283	Sw_GetStructuredGi()	330
Sw_DeallocateSessionProps()	284	Sw_GetStructuredText()	332
Sw_DeallocateTextItems()	285	Sw_IsExportingToXml()	334
Sw_EventHandler()	286	Sw_IsGeneralEntityDefined()	335
Sw_GetAssociatedEvent()	288	Sw_IsGeneralEntityNameUsed()	336
Sw_GetAttrVal()	290	Sw_NotifyEndTag()	337
Sw_GetAttrVals()	293	Sw_NotifyGeneralEntityDef()	338
Sw_GetBookCompEntityFilePath()	295	Sw_NotifyStartTag()	339
Sw_GetBookCompPi()	296	Sw_ScanElem()	340
Sw_GetBookPi()	297	Sw_SetAttrVal()	345
Sw_GetChildConvObj()	298	Sw_SetAttrVals()	347
Sw_GetColSpecs()	300	Sw_SetBookCompEntityFilePath()	350
Sw_GetCurConvObj()	302	Sw_SetBookCompPi()	351
Sw_GetCurConvObjOfType()	303	Sw_SetBookPi()	352
Sw_GetDocId()	304	Sw_SetColSpecs()	354
Sw_GetEntityName()	306	Sw_SetEntityName()	356
Sw_GetExportFileFormat()	307	Sw_SetExportFileFormat()	357

Sw_SetExportFilePath()	359	Sw_SetProcessingFlags().....	369
Sw_SetGfxDataAttrVals()	360	Sw_SetSessionProps()	370
Sw_SetGfxEntityName().....	361	Sw_SetStructuredGi()	372
Sw_SetGfxEntityType()	362	Sw_SetStructuredText().....	373
Sw_SetGfxNotation().....	364	Sw_SetTableScanOrder().....	374
Sw_SetGfxPubld()	365	Sw_WriteAttrSpec().....	376
Sw_SetGfxSysId()	366	Sw_WriteDelimiter()	377
Sw_SetPI()	367	Sw_WriteReservedName()	379
Sw_SetPrivateData()	368	Sw_WriteString()	380

Chapter 5 *Data Types and Structures Reference*

Primitive data types	383	SwTextItemTypeT	408
Enumerated data types	383	Data structures	409
StructuredAttrDeclaredValueT	384	FilePathT	409
StructuredAttrDefaultValueT	385	F_AttributeT	410
StructuredContentTokenT	385	F_AttributesT	410
StructuredContentTypeT	386	StructuredAttrDefT	411
StructuredDelimiterTypeT	387	StructuredAttrDefsT	411
StructuredEntityScopeT	388	StructuredAttrValT	412
StructuredEntityTypeT	389	StructuredAttrValsT	413
StructuredFeatureTypeT	390	StructuredElementDefT	413
StructuredQuantityTypeT	391	StructuredEntityDefT	414
StructuredReservedNameT	392	StructuredNotationDefT	415
SrEventTypeT	393	SrConvObjT	415
SrLocationT	394	SrEventT	416
SrObjTypeT	395	SrInsertLocT	416
SrwErrorT	396	SrSessionPropsT	417
SrwFmPropertyT	397	SrTagT	417
SrwFmPropValT	401	SRW_erno	418
SrwLogDocT	402	SrwColSpecT	418
SrwLogLocT	403	SrwColSpecsT	419
SrwTablePartTypeT	403	SrwLogMessageLocationT	420
SwEventTypeT	404	SrwPropValT	420
SwLocationT	406	SrwPropValsT	421
SwObjTypeT	407	SrwSpanSpecT	422

SrwSpanSpecsT	423	SwEventT	425
SrwStraddleT	423	SwSessionPropsT	425
SrwStraddlesT	424	SwTextItemT	426
SwConvObjT	424	SwTextItemsT	426
		SwTextItemsT	432

1

Introduction

The *Structure Import/Export API Programmer's Guide* is both a programming guide for coding, compiling, building and registering structure import and export clients, and a reference guide for the Adobe® FrameMaker® structure import/export API, which is part of the Frame® Developer's Kit (FDK) . This book is organized as follows:

- This introduction describes the intended audience for this information, and introduces the concepts underlying the creation of custom structure import/export clients.
- [Chapter 2, "Creating an Import/Export Client."](#) explains the basic coding needed to create a structure import/export client, how to register a client once it is built, and how to make the client available to end users of FrameMaker.
- [Chapter 3, "A Sample Import/Export Client."](#) provides details of the sample client code from the DocBook Starter Kit as an illustration of a working client.
- [Chapter 4, "Structure Import/Export API Function Reference."](#) provides a complete alphabetic reference for the structure import/export API function calls.
- [Chapter 5, "Data Types and Structures Reference."](#) provides an alphabetic reference for the data types, typedefs, and data structures used by the structure import/export API function calls

Who should read this guide?

The *Structure Import/Export API Programmer's Guide* is intended for developers who are knowledgeable C programmers and who want to build custom FDK clients for importing and exporting markup (XML or SGML) documents using FrameMaker. It assumes that you need import and export functionality that exceeds FrameMaker's built-in capabilities, including its read/write rules.

To use the *Structure Import/Export API Programmer's Guide* and write custom structure import and export clients for FrameMaker, you should have the following skills and knowledge:

- C or C++ programming experience.
- Working knowledge of XML or SGML.
- Familiarity with the *FrameMaker Structure Application Developer's Guide*, and experience setting up structured applications within FrameMaker.

- Familiarity with the Frame Developer's Kit (FDK) and Frame Development Environment (FDE). For more information about the FDK and FDE, see the *FDK Programmer's Guide* and the *FDK Programmer's Reference*.

What is a structure import/export client?

A structure import/export client is a program that enables you to control the import of markup documents into FrameMaker and the export of FrameMaker documents to markup. You code a structure import/export client in C, using the standard FDK libraries, described in the *FDK Programmer's Reference*, and a supplemental library for structured markup, described in this document. The markup library consists of the functions you call to control the import of markup documents to FrameMaker and the export of FrameMaker documents to markup.

Every time you import or export markup with FrameMaker, the process uses a structure import/export client. It uses either a custom client the user has specified in the Structured Application file, or the default client as specified in the Defaults section of the Structured Application file. (The Structured Application file is named `structapps.fm`. For more information about setting up structured applications, see the *FrameMaker Structure Application Developer's Guide*.)

A custom structure import/export client is an API client that you write, register with FrameMaker, and associate with a particular structured application through the Structured Application file. To add an entry for your custom client to the Structured Application file, see "Defining a structure import/export client for end users" on page 27.

Overview of the import and export process

Each time a user imports or exports markup, FrameMaker does the following:

1. Chooses the appropriate structured application and determines which client to use; either a custom client or the default client.
2. Initializes the appropriate client.
3. Parses the source document's content and structure, and formatting if applicable.
4. Builds a set of intermediate event and conversion object pairs, one at a time, describing what it proposes to write to the target document.
5. Passes each event/object pair, one at a time, to the current client.
6. When the client returns each event/conversion object pair, FrameMaker uses the pair to generate content or structure in the target document.

The import/export client can examine each event/conversion object and either return it as-is to FrameMaker, modify it and then return it, write directly to the target document, or drop the event/conversion object pair altogether.

FmTranslator: A minimal import/export client

The minimal client is the one provided as the default when you install FrameMaker. This client is named `FmTranslator`. It doesn't modify data for import or export; it simply returns unmodified event/conversion object pairs to FrameMaker. You will understand more about the import and export process when you finish this chapter, but seeing an example here may be helpful. The following code shows the event handlers in `FmTranslator`.

```
. . .
SrErrorT Sr_EventHandler(SrEventT *eventp, SrConvObjT srObj)
{
    return Sr_Convert (eventp, srObj);
}
SrErrorT Sw_EventHandler(SwEventT *eventp, SwConvObjT swObj)
{
    return Sw_Convert(eventp, swObj);
}
```

The client has two event handlers; `Sr_EventHandler()` for import and `Sw_EventHandler()` for export. FrameMaker passes each event and conversion object pair to the appropriate event handler as arguments. In the case of `FmTranslator`, the client simply returns every pair to FrameMaker — via `Sr_Convert()` for import, and via `Sw_Convert()` for export. When the client is done processing the final event/conversion object pair, the client passes control back to FrameMaker.

When to develop a structure import/export client

A custom structure import/export client is not always necessary. You should first try to accomplish what you need using default FrameMaker import and export, or modifications you can make with read/write rules. These methods are generally easier and less expensive to develop and maintain than clients.

However, you will sometimes find you cannot accomplish what you need using read/write rules and EDD modifications. This is when you need to develop a structure import/export client. Before you write a client, you should already have:

- A DTD
- A set of read/write rules
- For importing markup, a template file, complete with EDD

Of course, developing a structure application is an iterative process, and you will often find yourself modifying one or more of these files as you go. But you should make a first pass at developing these files before you begin developing your client. The client will operate in relation to these other files, and you may save effort by using solutions that rely on combining the effects of these files with your client.

For example, assume you need to track section levels and save them as attributes in XML; level 1 for the top level sections, level 2 for children of level 1 sections, and so on. For

export, you could have your client calculate the level of each section and write that value to an attribute in your XML. However, the EDD for your FrameMaker document can include statements to automatically track nesting levels. If you modify the EDD to accomplish this, then you can get the nesting level from the FrameMaker element and write its value to the attribute on export. By modifying the EDD, you spare yourself code in the import/export client.

For information about developing read/write rules and EDD's, see the *FrameMaker Structure Application Developer's Guide*. For an example of a structure application that uses read/write rules and a client to change import and export behavior, see the DocBook Starter Kit included with FrameMaker.

Examples of tasks requiring clients

FrameMaker provides a robust set of standard structure import and export behaviors, and allows you a great deal of additional control through read/write rules files. There are some import and export tasks, however, for which a structure import/export client is necessary. This section describes some known tasks that require a client.

- Column-major tables. If your DTD describes tables in column-major order, you need a client to handle importing and exporting of tables. The FrameMaker model for tables assumes row-major ordering.
- Processing instructions. If your markup document contains processing instructions, you need a client to perform that processing on import.
- Markup minimization (SGML, only). If you need to export markup minimization, you need to write a client to handle it.
- Changing the order of elements on import or export.

You may discover other import or export tasks that require clients. For example, the DocBook Starter Kit included with FrameMaker provides a client that includes the following capabilities:

- Strips extra white-space characters in text on import, except when the extra white-space occurs within `LiteralLayout` or `Screen` elements.
- Strips spaces following `LiteralLayout` or `Screen` elements on import.
- Strips leading spaces in a `Para` element on import.
- Unwraps `ULink` elements and creates text insets for them on import.
- Makes sure the `Mark` attribute for `ItemizedList` elements and the `Override` attribute for `Listitem` elements are mapped to one of four specific values on import.
- Maps `IndexTerm` subelements to the FrameMaker marker syntax.
- Maps FrameMaker text insets to `ULink` elements on export.

Understanding the import/export process

To write an effective import/export client, you need to understand the import/export process in some detail. You already know that import and export of markup is controlled by the current structure application, and that each application is defined in the Structured Application file. FrameMaker chooses an application on import according to the XML or SGML document type and stores it as a property in the resulting FrameMaker files; on export of a book or document FrameMaker uses this property to choose the structure application.

Each structure Application specifies which client to use. It can also specify:

- DOCTYPE
- DTD
- SGML Declaration file (for SGML, only)
- Character encoding (ANSI, ASCII, ISO Latin1, JIS, ShiftJIS, etc.) for import and export
- Entity locations
- Read/write rules file for import and export
- Template file
- Additional information

For more about structure applications, see the *FrameMaker Structure Application Developer's Guide*.

You should also know that to import or export markup, FrameMaker scans the source document and builds a series of event/conversion object pairs that it then passes to your client one at a time. An event contains information on what was encountered within the source document. A conversion object contains content related to that event. A structure import/export client relies on both pieces of information to determine what actions to take.

To perform import or export of markup, FrameMaker must:

1. Parse the read/write rules file associated with the application, and report any errors.

This occurs before FrameMaker posts any conversion events. If a read/write rules file is not specified for the current structure application, this step is skipped.

2. Begin the import (read) or export (write) process.

3. Parse the source document and post conversion events.

An event is a data structure that registers something of significance in either the source document or in the process itself. The list of import events is different from the list of export events. The difference is there because import events are based on XML or SGML content, and export events are based on FrameMaker document content. The events are enumerated in `SrEventType` and `SwEventType`. For complete descriptions of

these enumerated types, see “SrEventTypeT” on page 330 and “SwEventT” on page 356.

The import events are:

Begin reader	Begin book component	Begin element	Record end
End reader	End book component	End element	Processing instruction
Begin book	Begin document	Begin entity	CDATA
End book	End document	End entity	

The export events are:

Begin writer	End footnote	Begin table row	Marker
End writer	Begin table	End table row	Cross-reference
Begin book	End table	Begin table cell	Graphic
End book	Begin table title	End table cell	Equation
Begin book component	End table title	Begin Rubi Group	Text
End book component	Begin table heading	End Rubi Group	Special character
Begin document	End table heading	Begin Rubi	Reference element
End document	Begin table body	End Rubi	Text inset
Begin element	End table body	Begin colspec	End of line
End element	Begin table footing	End colspec	End of paragraph
Begin footnote	End table footing	Variable	Condition change

4.For each event, FrameMaker constructs a proposed conversion object.

If the current structure application includes read/write rules, FrameMaker uses those rules when building the proposal.

5.FrameMaker passes the event and proposed conversion object to the current import/export client.

The client can:

- Return the event/conversion object pair directly to FrameMaker without modifying it.
- Drop the event/conversion object pair entirely.
- Drop the content of the event/conversion object pair, write content directly into the target document, and return the event/conversion object pair to FrameMaker.
- Modify the conversion object and then return the event/conversion object pair to FrameMaker.

6.For each conversion object the client returns, FrameMaker uses that conversion object to generate content or structure in the target document.

7.For each event the client returns, FrameMaker manages the event’s place on the event stack.

The event stack maintains a hierarchy of open events. When the import/export client returns an *open* event, FrameMaker puts the event on the stack. When the client returns a *close* event, FrameMaker removes the corresponding *open* event from the stack. In this way, the events track document hierarchy.

For example, assume a Section element contains many Paragraph elements. The Section event will remain open until its last Paragraph element has been processed. Also note, no two sibling events can be open at the same time; only parent events and their descendents can be open at a given time.

The structure import/export API provides calls to get open parent and child events from the stack.

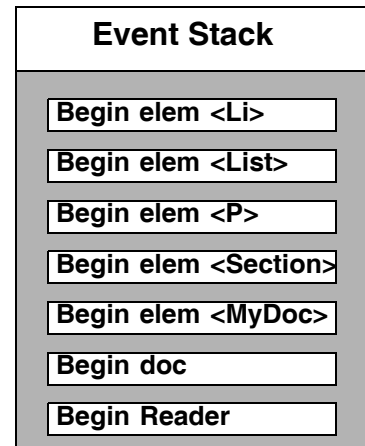
8. After the client returns the last event of an import process, FrameMaker applies formatting to the target document according to the newly created document's EDD.

On import, the DTD is part of the markup document instance. For SGML, the SGML declaration may also be part of the document instance. Otherwise, the location of the SGML declaration is specified in `structapps.fm`. However, if you import an XML or SGML fragment, the DTD is not a part of that fragment. In that case FrameMaker uses the DTD that is specified for the current structure application.

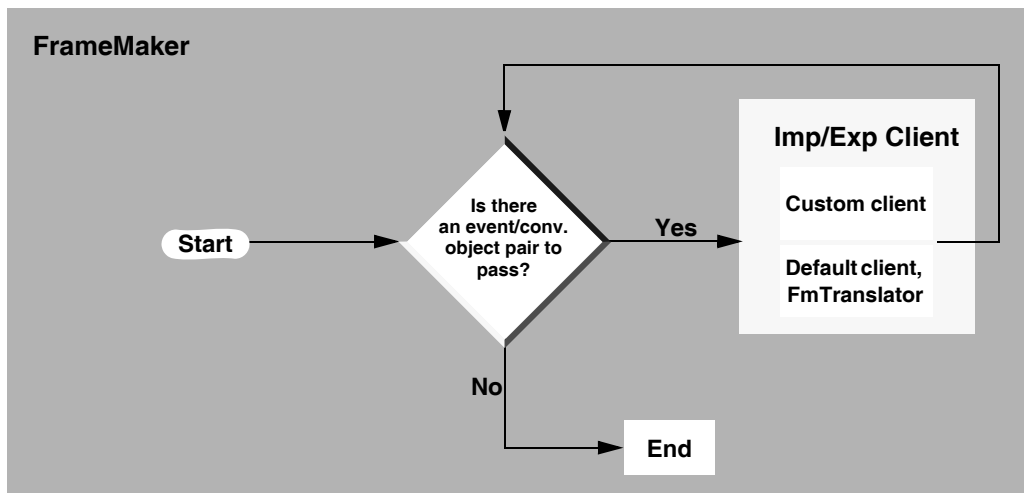
On export, FrameMaker uses the DTD and (for SGML) the SGML declaration that are specified in `structapps.fm`.

The following diagram illustrates the way structure in a markup document corresponds to events, and how the event stack maintains a list of currently open events. The shaded entries for the markup document and corresponding events are entries for events that have been removed from the event stack. This is a simplified illustration. In reality there would be more events for each element; for example, this diagram does not account for the PCDATA content of each paragraph.

XML Document	Corresponding Event
Import XML	Begin Reader
	Begin doc
<MyDoc>	Begin elem
<Section>	Begin elem
<P>	Begin elem
</P>	Ended elem
</Section>	Ended elem
<Section>	Begin elem
<P>	Begin elem
</P>	Ended elem
<P>	Begin elem
<List>	Begin elem
	Begin elem



The following flow chart illustrates how FrameMaker and the current structure application's import/export client interact to convert event/conversion objects from the stack.



Processing events based on their type

A structure import/export client is nothing more than an event handler; it never changes events, it responds to them. For all the events that arise in an import or export process, a subset of these events might need custom processing. The import/export client identifies

which events to process, and modifies the conversion objects for those events. It then returns to FrameMaker every event/conversion object pair that will yield the desired content or structure in the target document.

To determine which events to process, the event handler examines the event type and processes the event accordingly. Typically, the handler includes a switch statement with cases based on the enumerated event types. To illustrate, here is a code fragment from the import routine for the import/export client for the DocBook Starter Kit. This example shows how the import process uses `Sr_EventHandler()` to trap specific events:

```
. . .
SrErrorT Sr_EventHandler(SrEventT *eventp, SrConvObjT srObj)
{
    /* Trap events by event-type here */
    switch (eventp->evtype)
    {
        case SR_EVT_BEGIN_READER:
            . . .
        case SR_EVT_BEGIN_ELEM:
            . . .
        case SR_EVT_END_ELEM:
            . . .
        case SR_EVT_CDATA:
        case SR_EVT_RE:
            . . .
        default:
            break;
    }
    /* Pass event/object to converter and return control. */
    return (Sr_Convert (eventp, srObj));
}
```

As this code fragment suggests, event handlers usually trap only a subset of possible events. However, you must take care to pass all unmodified event/conversion object pairs to the appropriate structure import/export API conversion routine (`Sr_Convert()` for import routines, and `Sw_Convert()` for export routines).

In the above example, the code returns all event/conversion object pairs after falling through the switch statement. If one of the case statements returns without falling through to `return(Sr_Convert(eventp, srObj))`, the corresponding event/conversion object pair will not get written to the target document. This is a viable way to exclude data from the target document. For example, a routine might write directly to the target document instead of calling a `Sr_Convert()`. In that case, the routine can trap the event, write to the target document, and then return `SRW_E_SUCCESS` (to indicate a successful conversion) to FrameMaker.

2

Creating an Import/Export Client

This chapter describes how to create structure import/export clients. You write the code in C, using the Frame Developer's Kit (FDK), the Frame Development Environment (FDE), and the structure import/export API library.

To introduce the elementary concepts of client design, this chapter uses the default structure import/export client, `FmTranslator`. It then describes the steps necessary for you to code, compile, build, register, and define a custom client for end users.

Sample code for `FmTranslator` is included with the FDK. You can find the sample code in your FDK installation directory: `.../fdk/samples/struct`.

Note that `FmTranslator` is the minimal client; it performs no modifications to the event/conversion object pairs it receives. It merely invokes the API conversion routines (`Sr_Convert()` for import and `Sw_Convert()` for export) to convert the pairs as is. Later on you will see examples of handlers that build on the fundamental structure of `FmTranslator`.

Understanding FmTranslator

The code for the default client, `FmTranslator`, illustrates the most elementary form an import/export client can take. Like `FmTranslator`, most clients contain routines for both import and export, and most clients also call the structure import/export API conversion routines to convert event/conversion object pairs. Your clients will share these features with `FmTranslator`.

The code for `FmTranslator` is divided into three modules; one for import events (`Rdrevent.c`), one for export events (`Wtrevent.c`), and one to call the structure import/export API callbacks from within the standard FDK callback routines (`Trnslate.c`).

FmTranslator import and export modules

The import module, `Rdrevent.c` consists of nothing more than the import event handler, `Sr_EventHandler()`:

```
#include "fm_struct.h"

SrwErrorT Sr_EventHandler(SrEventT *eventp, SrConvObjT srObj)
{
    return Sr_Convert (eventp, srObj);
}
```

Likewise, the export module, `Wtrevent.c` consists of nothing more than the export event handler, `Sw_EventHandler()`:

```
#include "fm_struct.h"

SrwErrorT Sw_EventHandler(SwEventT *eventp, SwConvObjT srObj)
{
    return Sw_Convert (eventp, swObj);
}
```

Each handler receives an event/conversion object pair as arguments, and then immediately returns the results of the appropriate conversion routine; `Sr_Convert()` for import, and `Sw_Convert()` for export. In this way, `FmTranslator` doesn't modify the import and export behavior that `FrameMaker` proposes.

According to the structure import/export API naming conventions, the `Sr` prefix is for import (reading) routines and data types, the `Sw` prefix is for export (writing) routines and data types, and the `Srw` prefix is for functions and data types that can be used in both import and export routines. In the above code, the prefix to `Sr_Convert()` indicates that it is an import routine. However, the import and export handlers both return the same data type, `SrwErrorT`. For more about the classes of structure import/export API functions, see [“Class of functions” on page 49](#).

Modifying FDK callbacks

`FmTranslator` includes the module `Trnslate.c`, which modifies the standard FDK callbacks to include the structure import/export API versions of these callbacks. The structure import/export API versions of the callbacks are opaque to you; you must add them to any FDK callbacks you will use in an import/export client.

The following table lists the FDK callbacks and their associated structure import/export API callbacks:

FDK callback function	Structured Import/Export callback function
<code>F_ApiCommand()</code>	<code>Structured_ApiCommand()</code>
<code>F_ApiDialogEvent()</code>	<code>Structured_ApiDialogEvent()</code>
<code>F_ApiEmergency()</code>	<code>Structured_ApiEmergency()</code>
<code>F_ApiInitialize()</code>	<code>Structured_ApiInitialize()</code>
<code>F_ApiNotify()</code>	<code>Structured_ApiNotify()</code>

As a minimum you must modify any callbacks you intend to use. However, it's just as well for you to include a module like `Trnslate.c` that modifies all the FDK callbacks. Then if you want to use a callback, you can add your code to it in that module.

Trnslate.c begins with prototypes for the structure import/export API versions of the callbacks. The actual callback routines then call the structure import/export API versions. Following is the code for Trnslate.c:

```
#include "fm_struct.h"
#include "futils.h"

extern VoidT Structured_ApiEmergency FARGS((VoidT));
extern VoidT Structured_ApiInitialize FARGS((IntT init));
extern VoidT Structured_ApiCommand FARGS((IntT command));
extern VoidT Structured_ApiNotify FARGS((IntT notification,
                                         F_ObjHandleT docId, StringT sparm,
                                         IntT iparm));
extern VoidT Structured_ApiDialogEvent FARGS((IntT dlgNum,
                                              IntT itemNum, IntT mods));

VoidT F_ApiEmergency() {
    Structured_ApiEmergency();
}

VoidT F_ApiInitialize(IntT init) {
    Structured_ApiInitialize(init);
}

VoidT F_ApiCommand(IntT command) {
    Structured_ApiCommand(command);
}

VoidT F_ApiNotify(IntT notification,
                  F_ObjHandleT docId, StringT sparm, IntT iparm) {
    Structured_ApiNotify(notification, docId, sparm, iparm);
}

VoidT F_ApiDialogEvent(IntT dlgNum, IntT itemNum, IntT mods) {
    Structured_ApiDialogEvent(dlgNum, itemNum, mods);
}
```

If you want to add code to one of the callbacks, just add it after the call to the structure import/export API routine. You can go ahead and code the callbacks as you normally would for an FDK client. For more information about the FDK callback routines, see the *FDK Programmer's Reference*.

Creating a custom import/export client

You might want to use the core code for FmTranslator as a coding template for all your custom clients. Custom clients are more complex and may involve additional code modules,

but `FmTranslator` lays a basic foundation for all clients, no matter how complex they become.

The next several sections of this chapter describe the following pieces of code you should include in your clients:

- Header files.
- Global declarations and constants, both optional, but frequently useful.
- Import and export event handlers; this section looks at them in more detail.
- Entity handlers.

Header files

The source code for all import/export clients must include the `fm_struct.h` header file. This header file includes header files needed to create a simple client. It provides typedefs, enums, data structures, and function prototypes used throughout the structure import/export API. The include statement for this header file should come first in any module that uses structure import/export API functions.

Most clients also use the portable memory libraries provided with the Frame Development Environment (FDE). You will also frequently use FDE string, string list, character, and I/O libraries. You should become familiar with the FDE; for example, by using FDE string and character handling, you can develop parsing routines that will work on every platform FrameMaker supports. For a complete discussion of the FDE and its libraries, see the *FDK Programmer's Guide*.

If you use any FDE calls in a client, you must also add the appropriate header files to the client source code. For example, if you use both FDE memory and string calls in a client, you need to add the following include statements to your code:

```
#include "fmemory.h"
#include "fstrings.h"
```

If you need to include any C library files, add the appropriate include statements after the structure import/export API and FDE header files.

Global declarations and constants

Most clients define constants and macros, and declare typedefs, global variables, and structures that are used by the functions that make up the client. You declare these as you would in any C program.

You need global and static variables if you want to store a value from one event so it can be used in another. All variables within an event handler are local to the handler, so their values do not persist from event to event. For example, if you want to keep a running count of the footnotes that pass through your handler, you would declare a global variable and increment it for each footnote event.

Event handlers

Event handlers usually consist of a `switch` statement triggered by the type of event associated with an event/conversion object pair. The following code fragment shows how to set up the `switch` statement for an import event handler:

```
. . .
SrErrorT Sr_EventHandler(SrEventT *eventp, SrConvObjT srObj)
{
    switch (eventp->evtype)
    {
        case SR_EVT_BEGIN_READER:
            . . .
        case SR_EVT_BEGIN_ELEM:
            . . .
        case SR_EVT_END_ELEM:
            . . .
        case SR_EVT_CDATA:
            . . .
        default:
            break;
    }
    return (Sr_Convert (eventp, srObj));
}
. . .
...

```

The above code shows the import event handler trapping event type through `eventp->evtype`. (With export handlers you can trap event type for export events.) The import event types are enumerated in `SrEventTypeT`, and the export events types are enumerated in `SwEventTypeT`. For complete descriptions of these enumerated types, see [“SrEventTypeT” on page 393](#) and [“SwEventTypeT” on page 404](#).

You can also set up an event handler to trap for conversion object types. While the event type tells you about what it was in the source document that triggered the event, the conversion object type tells you about FrameMaker document objects. For import, it tells you about the document object FrameMaker proposes to create in the target document; for export it tells you about the document object in the target FrameMaker document that triggered the event. Conversion object types are enumerated in `SrObjTypeT` for import objects, and `SwObjTypeT` for export objects. For more information, see [“SrObjTypeT” on page 395](#) and [“SwObjTypeT” on page 407](#).

The following code shows an import event handler that traps *element* types — this fragment shows cases for graphics and equations. The example uses a `switch` statement to illustrate

that you can check for many other types of conversion object types. If the conversion event is a begin element event, the code saves the object ID in a variable.

```
. . .
F_ObjHandleT graphicId, eqnId; /* global vars */
. . .
SrErrorT Sr_EventHandler(SrEventT *eventp, SrConvObjT srObj)
{
    SrObjTypeT objType;

    objType = Sr_GetObjType(srObj);
    switch (objType)
    {
        case SR_OBJ_GRAPHIC:
            if(eventp->evtype == SR_EVT_BEGIN_ELEM) {
                Sr_Convert(eventp, srObj);
                graphicId = Sr_GetFmObjId(srObj);
                return(SRW_E_SUCCESS);
            }
            break;
        case SR_OBJ_EQUATION:
            if(eventp->evtype == SR_EVT_BEGIN_ELEM) {
                Sr_Convert(eventp, srObj);
                eqnId = Sr_GetFmObjId(srObj);
                return(SRW_E_SUCCESS);
            }
            break;
        . . .
        default:
            break;

    } /* End switch on objType */
    . . .
    return (Sr_Convert (eventp, srObj));
}
```

Each case statement that converts an event/object pair should return directly; if it falls through to call `Sr_Convert()` outside of the switch statement, then the event/conversion pair will be converted twice.

When importing markup, the corresponding FrameMaker document objects don't exist until after calling `Sr_Convert()`. This means you can't get the ID of an actual FrameMaker object until after the corresponding XML or SGML has been converted.

Entity handlers

You use `Srw_EntityHandler()` to customize entities that refer to external files in ways the read/write rules do not support. For example, you can use it to generate external files at import time. Assume an entity declaration that specifies a query file; the handler can read that query and pass it to a database. After the database processes the query, the handler can return the file path of the query result.

`Srw_EntityHandler()` receives the entity name, the entity scope, and the proposed file path as arguments. You use the entity name to trap the entity you want to modify; use entity name and entity scope to get the full entity declaration.

You use the file path argument to open and read the entity's data file. Then you can use that data to generate the file for your document content. Once you generate the content file, you return its file path to FrameMaker. The result is a text inset displaying the content file in your FrameMaker document.

The text inset properties show the entity's data file as the source for the text inset. Thus, when you export, FrameMaker can identify the entity and write out the correct entity reference.

To follow the code structure in the example client, you can create a separate module for the entity handler. For example, if you create a file named `Entity.c` and add it to the sample structure import/export API project (found in your FDK directory: `.../FDK/Samples/Struct`), you can then build an import/export client that includes an entity handler.

Following is an example of `Srw_EntityHandler()` that:

- Checks the `entName` argument for a specific entity reference.
- Opens and reads the entity's data file.
- Checks the system to see if the handler already created content file for that entity.
- If a file already exists, returns that file path. This is important if you want to ensure a single process created the source for each entity reference.
- If no file exists, runs a process to generate that file. In this case, the code displays a dialog box prompting you for a string. It then writes the string to the content file.

- If it could create and write to the file, returns the file path; otherwise, returns NULL.

```
#include "fm_struct.h"
#include "futils.h"
#include "fdetypes.h"
#include "fstrings.h"
#include "fchannel.h"
#define BUFFERSIZE (IntT)256
FilePathT *Srw_EntityHandler(StringT entName,
                             StructuredEntityScopeT scope, FilePathT *defaultFp)
{
    const StructuredEntityDefT *entDef;
    StringT s;
    ChannelT chan;
    UCharT ptr[BUFFERSIZE];
    IntT err;

    if(F_StrEqual(entName, (StringT)"query1") {
/* Get the contents of the data file */
        if((chan = F_ChannelOpen(defaultFp,"r")) != NULL) {
            numread = F_ChannelRead(ptr, sizeof(UCharT),
                                    BUFFERSIZE-1, chan);
            ptr[numread] = '\0';
            F_ChannelClose(chan);
        }
/* Change defaultFp to the file the handler will create */
        defaultFp = F_PathNameToFilePath(
            (StringT)"<r><c>temp<c>query1.txt", NULL, FDIPath);
/* If content file exists, return path. */
        if((chan = F_ChannelOpen(defaultFp,"r")) != NULL) {
            F_ChannelClose(chan);
            return(defaultFp);
        }
/* Otherwise, generate content file - should warn */
/* the user if F_ChannelOpen() fails. In this example, */
/* the processing is just getting content into string s. */
        err = F_ApiPromptString(&s, "String?", ptr);
        if((chan = F_ChannelOpen(defaultFp,"w")) == NULL)
            return(NULL);
        F_ChannelWrite(s, sizeof(UCharT), F_StrLen(s), chan);
        F_ChannelClose(chan);
        F_ApiDeallocateString(&s);
    }
/* If you don't modify the entity, this returns */
/* the file path unchanged. Otherwise, returns */
```

```
/* the path to the content file. */
return(defaultFp);
}
```

With the above entity handler, if you import the following markup, the handler should prompt you once for a string. The string you supply should appear twice in the resulting FrameMaker document.

```
<!DOCTYPE test [
  <!ELEMENT test      - -  (#PCDATA) >
  <!ENTITY query1 SYSTEM "C:\queries\MyQuery1.sql">
]>
<test>
  Here is a reference to query1: &query1;
  Here is another reference to query1: &query1;
</test>
```

Note the prefix for this handler is `Srw_`, which means your client uses the same handler for import and export of markup. On export, the handler returns the same file path you specified for import; in the above example, `defaultFp` specifies the same content file for both import and export.

However, the value of the system or public ID is the same as in the original entity declaration. In the above example, the system ID for import and export is `C:\queries\MyQuery1.sql`.

Following is an example that shows how to print the public and system ID to the console for every external entity you import or export. It also shows how to use `entName` and `scope` to get the entity declaration. Note that this example never modifies the entities; it always returns the unmodified file path.

```
#include "fm_struct.h"
#include "futils.h"

FilePathT *Srw_EntityHandler(StringT entName,
                             StructuredEntityScopeT scope, FilePathT *defaultFp)
{
    const StructuredEntityDefT *entDef;

    entDef = Structured_GetEntityDef(scope, entName);
    if(!F_StrIsEmpty(entDef->pubid))
        F_Printf(NULL, "\n pubid: %s", entDef->pubid);
    if(!F_StrIsEmpty(entDef->sysid))
        F_Printf(NULL, "\n sysid: %s", entDef->sysid);
    return(defaultFp);
}
```

It is important to understand that `Srw_EntityHandler()` does not create text insets for entities that refer to FrameMaker files. To accomplish that you will have to use the FDK to import the content of the FrameMaker document that is indicated by `defaultFp`.

In the markup example above, the entity declaration specifies an external text file. The handler also supports graphics when the graphic element uses an entity attribute to specify the graphic file.

Compiling, building, and registering a client

You compile, build, and register an import/export client very much as you do any FDK client. Instructions for compiling, linking, and registering are specific to each platform. For information, see the *FDK Platform Guide* for your platform.

Defining a structure import/export client for end users

To make an import/export client available to end users, you must define a structure application in the Structure Application file, `structapps.fm`. The `UseApiClient` entry for the application definition specifies which client to use. You type the registered name of your client in this entry.

A structure import/export client is just one part of a structure application. In order to take full advantage of FrameMaker, you should become familiar with all parts of structure applications. You should also know how to edit the Structure Application file. For a complete discussion of structure applications and editing the application file, see the *FrameMaker Structure Application Developer's Guide*.

3

A Sample Import/Export Client

This chapter uses the XHTML Starter Kit import/export client as an example to explain how structure import/export API functions interact in clients. The example in this chapter also shows how clients can combine standard Frame Developer's Kit (FDK) and Frame Development Environment (FDE) functions with the structure import/export API calls to accomplish conversion tasks. The example includes an overview of this XHTML import and export client, shows the code for the import and export event handlers, and describes what the code does.

Your installation of FrameMaker also includes versions of the DocBook Starter Kit—one for SGML(version 4.1) and another for XML (version 4.1.2). These starter kits use import/export clients, and you can find the source code for the clients alongside the starter kit installation.

Overview of the sample XHTML client

This client represents one of many pieces of a structure application. The XHTML Starter Kit that ships with FrameMaker is a functioning structure application designed to import XML documents that conform to the XHTML DTD into FrameMaker, and export FrameMaker documents to XHTML-compliant XML. You should look at the complete XHTML Starter Kit to see how the import/export client, the application definition, the read/write rules, and all the other files in the application work together.

The example code described here is an example of an XHTML client. You can use it as is, or as a starting point for implementing more XHTML import/export functionality. The XHTML client is, however, only one part of the Starter Kit. The Starter Kit also includes the XHTML DTD, a FrameMaker EDD, and a read/write rules file, and other data to use in a structure application. The XHTML client works in conjunction with these other parts of the Starter Kit.

This example client is designed for import and export of XHTML, and it performs the following tasks:

- Manages the import of tables. To create a table in FrameMaker, you must already know the number of columns in the table. However, XHTML doesn't require a specification for the number of columns. The client creates a 100-column table, imports the XHTML table rows and cells, then deletes empty columns.
- Converts `<a>` elements to hypertext newlink or gotolink markers.
- Imports forms elements and script elements as conditional text so you can hide them for printing and viewing the document, and show them when saving as XHTML.

Source code for the XHTML client is split into three modules. `import.c` contains all code for importing from XHTML; `export.c` contains the code for exporting from FrameMaker to XHTML; `translate.c` contains initialization code, and is not discussed here.

XHTML import

For the import of XHTML the starter kit uses eight functions:

- `SetWidthOnImport()` to set the table and column widths when importing a table
- `CenterOnImport()` to apply a centered alignment for table elements
- `DeleteCols()` to delete empty table columns
- `UpdateTblBorders()` to apply specified borders to tables on import
- `InsertMarker()` to create hypertext markers for `<a>` elements
- `getHypertextMarker()` a convenience function to get the correct marker type
- `getAttrVal()` a convenience function to get the value of the specified attribute

These functions are implemented in the import module, but this example does not show their implementation. You can see these functions in the source code that is included in the XHTML starter kit.

```
1  #include "fm_struct.h"
2  #include "fstrings.h"
3  #include "fstrlist.h"
4  #include "fmemory.h"
5  #include "futils.h"
6  #include "fcharmap.h"
7  #include "fmetrics.h"
8  #include "fstrres.h"
9
10 /*
11  * function prototypes
12  */
13 SrwErrorT InsertMarker(F_ObjHandleT docId, F_ObjHandleT
elemId);
14 VoidT SetWidthOnImport(F_ObjHandleT docId, F_ObjHandleT
elemId);
15 VoidT CenterOnImport(F_ObjHandleT docId, F_ObjHandleT elemId);
16 StringT getAttrVal(F_ObjHandleT docId, F_ObjHandleT elemId,
StringT attrname);
17 VoidT ApplyCondition(F_ObjHandleT docId, F_ObjHandleT elemId,
18                     StringT condition);
19 VoidT DeleteCols(F_ObjHandleT docId, F_ObjHandleT elemId);
20 VoidT UpdateTblBorders(F_ObjHandleT docId, F_ObjHandleT elemId,
21                       SrConvObjT tblObj);
22 F_ObjHandleT getHypertextMarkerType(F_ObjHandleT docId);
23
24 /* macros */
25 #define NUM_ELEMENTS(x) (sizeof(x)/sizeof((x)[0]))
26 #define BUFF 255
27
28 /* FrameMaker metrics */
29 #define pts (MetricT) 65536
30 #define in (MetricT) 65536 * 72
31
32 /* nested table counter */
33 static IntT tablecount = 0;
34
35 /* cell counter */
36 static IntT cellcount = 0;
37 static IntT totalcells = 0;
38
```

```
39  /* total number of columns in a table. see rules document to
    change.*/
40  static IntT columnTotal = 100;
41
42  /* widths of table cells */
43  static MetricT tblWidths[100];
44  static BoolT USECELLWIDTHS = True;
45
46  static SrInsertLocT lastInsertLoc;
47  static SrInsertLocT lastCellInsertLoc;
48  static F_ObjHandleT docId;
49  static SrConvObjT docObj;
50
51  /* defines for align attribute */
52  #define CENTER  1
53  #define RIGHT   2
54  #define LEFT    3
55
56  /*
57   * XML Reader Sample Event Handler for XHTML
58   */
59  SrErrorT Sr_EventHandler(eventp, srObj)
60  SrEventT *eventp;
61  SrConvObjT srObj;
62  {
63      F_ObjHandleT elemId;
64      static BoolT TD = True;
65      static BoolT TH = True;
66      static IntT widthcellcount = 0;
67      static IntT tablerowcount = 0;
68      F_AttributeT cellWidthAtt;
69
70      switch (eventp->evtype)
71      {
72      case SR_EVT_BEGIN_READER:
73      case SR_EVT_BEGIN_BOOK:
74      case SR_EVT_BEGIN_DOC:
75      case SR_EVT_BEGIN_BOOK_COMP:
76          docObj = Sr_GetCurConvObjOfType(SR_OBJ_DOC);
77          docId = Sr_GetDocId(docObj);
78          break;
79
80      case SR_EVT_CDATA :
81          break;
```



```
82
83     case SR_EVT_BEGIN_ELEM:
84         if (F_StrIEqual(eventp->u.tag.gi, (StringT)"table"))
85         {
86             tablecount++;
87             if (tablecount > 1)
88                 return (SRW_E_SUCCESS);
89         }
90     else if (F_StrIEqual(eventp->u.tag.gi, (StringT)"tr"))
91     {
92         if (tablecount == 1)
93             tablerowcount++;
94
95         if (tablecount > 1)
96             return (SRW_E_SUCCESS);
97     }
98 }
99 else if (F_StrIEqual(eventp->u.tag.gi, (StringT)"td"))
100 {
101     if (tablecount == 1)
102     {
103         lastInsertLoc = Sr_GetInsertLoc(srObj);
104         TD = True;
105         lastCellInsertLoc = Sr_GetInsertLoc(srObj);
106
107         /* get width of cell */
108         if (tablerowcount == 1)
109         {
110             cellWidthAtt = Sr_GetAttrVal(srObj,
111 (StringT)"width");
112             if (cellWidthAtt.values.len > 0)
113             {
114                 tblWidths[widthcellcount] = F_StrAlphaToInt(
115                     (StringT)cellWidthAtt.values.val[0])
116 * pts;
117                 F_ApiDeallocateAttribute(&cellWidthAtt);
118                 widthcellcount++;
119             }
120             else
121                 USECELLWIDTHS = False;
122         }
123     }
124     else if (tablecount > 1)
125     {
```

```
124         Sr_SetInsertLoc(srObj, &lastInsertLoc);
125         if (TH == True)
126             Sr_SetFmElemTag(srObj, (StringT) "th");
127         Sr_SetProcessingFlags(srObj, SRW_UNWRAP_ELEMENT);
128     }
129 }
130 else if (F_StrIEqual(eventp->u.tag.gi, (StringT)"th"))
131 {
132     if (tablecount == 1)
133     {
134         lastInsertLoc = Sr_GetInsertLoc(srObj);
135         TH = True;
136         lastCellInsertLoc = Sr_GetInsertLoc(srObj);
137
138         /* get width of cell */
139         if (tablerowcount == 1)
140         {
141             cellWidthAtt = Sr_GetAttrVal(srObj,
142 (StringT)"width");
143             if (cellWidthAtt.values.len > 0)
144             {
145                 tblWidths[widthcellcount] = F_StrAlphaToInt(
146                     (StringT)cellWidthAtt.values.val[0])
147 * pts;
148                 F_ApiDeallocateAttribute(&cellWidthAtt);
149                 widthcellcount++;
150             }
151             else
152                 USECELLWIDTHS = False;
153         }
154     }
155     else if (tablecount > 1)
156     {
157         Sr_SetInsertLoc(srObj, &lastInsertLoc);
158         if (TD == True)
159             Sr_SetFmElemTag(srObj, (StringT) "td");
160         Sr_SetProcessingFlags(srObj, SRW_UNWRAP_ELEMENT);
161     }
162 }
163 else if (F_StrIEqual(eventp->u.tag.gi, (StringT)"tfoot"))
164 {
165     if (tablecount > 1)
166     {
167         Sr_SetProcessingFlags(srObj, SRW_UNWRAP_ELEMENT);
```

```
166         return (SRW_E_SUCCESS);
167     }
168 }
169 else if (F_StrIEqual(eventp->u.tag.gi, (StringT)"thead"))
170 {
171     if (tablecount > 1)
172     {
173         Sr_SetProcessingFlags(srObj, SRW_UNWRAP_ELEMENT);
174         return (SRW_E_SUCCESS);
175     }
176 }
177 else if (F_StrIEqual(eventp->u.tag.gi,
178 (StringT)"caption"))
179 {
180     if (tablecount > 1)
181     {
182         Sr_SetProcessingFlags(srObj, SRW_UNWRAP_ELEMENT);
183         return (SRW_E_SUCCESS);
184     }
185 }
186 else if (F_StrIEqual(eventp->u.tag.gi, (StringT)"tbody"))
187 {
188     if (tablecount > 1)
189     {
190         Sr_SetProcessingFlags(srObj, SRW_UNWRAP_ELEMENT);
191         return (SRW_E_SUCCESS);
192     }
193 }
194 /* Insert hypertext marker for "a" element. */
195 else if (F_StrIEqual(eventp->u.tag.gi, (StringT)"a"))
196 {
197     Sr_Convert(eventp, srObj);
198
199     docObj = Sr_GetCurConvObjOfType(SR_OBJ_DOC);
200     docId = Sr_GetDocId(docObj);
201     elemId = Sr_GetFmElemId(srObj);
202     return InsertMarker(docId, elemId);
203 }
204 }
205 break;
206
207 /
*****
```

```
*****
208         END ELEMENT EVENT
209         *****/
*****/
210     case SR_EVT_END_ELEM:
211         /*
212         *   tablecount needs to be decremented.
213         */
214         if (F_StrIEqual(eventp->u.tag.gi, (StringT)"table"))
215         {
216             if (tablecount == 1)
217             {
218                 docObj = Sr_GetCurConvObjOfType(SR_OBJ_DOC);
219                 docId = Sr_GetDocId(docObj);
220                 elemId = Sr_GetFmElemId(srObj);
221                 DeleteCols(docId, elemId);
222                 Sr_Convert(eventp, srObj);
223                 SetWidthOnImport(docId, elemId);
224                 CenterOnImport(docId, elemId);
225                 UpdateTblBorders(docId, elemId, srObj);
226                 cellcount = 0;
227                 totalcells = 0;
228                 tablecount--;
229                 if (tablecount == 0)
230                 {
231                     USECELLWIDTHS = True;
232                     tablerowcount = 0;
233                     widthcellcount = 0;
234                 }
235                 return (SRW_E_SUCCESS);
236             }
237             else
238             {
239                 Sr_SetProcessingFlags(srObj, SRW_UNWRAP_ELEMENT);
240                 tablecount--;
241                 return Sr_Convert(eventp, srObj);
242             }
243         }
244         else if (F_StrIEqual(eventp->u.tag.gi, (StringT)"tr"))
245         {
246             if (tablecount == 1)
247             {
248                 if (cellcount > totalcells)
249                     totalcells = cellcount;
```

```
250
251         cellcount = 0;
252     }
253     if (tablecount > 1)
254     {
255         Sr_SetProcessingFlags(srObj, SRW_UNWRAP_ELEMENT);
256     }
257 }
258 else if (F_StrIEqual(eventp->u.tag.gi, (StringT)"td"))
259 {
260     if (tablecount == 1)
261     {
262         Sr_Convert(eventp, srObj);
263         lastInsertLoc = Sr_GetInsertLoc(srObj);
264         cellcount++;
265         return (SRW_E_SUCCESS);
266     }
267     Sr_SetProcessingFlags(srObj, SRW_UNWRAP_ELEMENT);
268 }
269 else if (F_StrIEqual(eventp->u.tag.gi, (StringT)"th"))
270 {
271     if (tablecount == 1)
272     {
273         Sr_Convert(eventp, srObj);
274         lastInsertLoc = Sr_GetInsertLoc(srObj);
275         cellcount++;
276         return (SRW_E_SUCCESS);
277     }
278     if (tablecount > 1)
279     {
280         Sr_SetProcessingFlags(srObj, SRW_UNWRAP_ELEMENT);
281     }
282 }
283 else if (F_StrIEqual(eventp->u.tag.gi, (StringT)"tfoot"))
284 {
285     if (tablecount > 1)
286     {
287         Sr_SetProcessingFlags(srObj, SRW_UNWRAP_ELEMENT);
288     }
289 else if (F_StrIEqual(eventp->u.tag.gi, (StringT)"thead"))
290 {
291     if (tablecount > 1)
292     {
293         Sr_SetProcessingFlags(srObj, SRW_UNWRAP_ELEMENT);
294     }
```

```
294         else if (F_StrIEqual(eventp->u.tag.gi,  
(StringT)"caption"))  
295         {  
296             if (tablecount > 1)  
297                 Sr_SetProcessingFlags(srObj, SRW_UNWRAP_ELEMENT);  
298         }  
299     else if (F_StrIEqual(eventp->u.tag.gi, (StringT)"tbody")  
300     {  
301         if (tablecount > 1)  
302             Sr_SetProcessingFlags(srObj, SRW_UNWRAP_ELEMENT);  
303     }  
304     else if (F_StrIEqual(eventp->u.tag.gi, (StringT)"label")  
||  
305         F_StrIEqual(eventp->u.tag.gi, (StringT)"select")||  
306         F_StrIEqual(eventp->u.tag.gi, (StringT)"optgroup")  
||  
307         F_StrIEqual(eventp->u.tag.gi, (StringT)"option")||  
308         F_StrIEqual(eventp->u.tag.gi, (StringT)"textarea")  
||  
309         F_StrIEqual(eventp->u.tag.gi, (StringT)"fieldset")  
||  
310         F_StrIEqual(eventp->u.tag.gi, (StringT)"button")||  
311         F_StrIEqual(eventp->u.tag.gi, (StringT)"legend")||  
312         F_StrIEqual(eventp->u.tag.gi, (StringT)"isindex")||  
313         F_StrIEqual(eventp->u.tag.gi, (StringT)"input"))  
314     {  
315         Sr_Convert(eventp, srObj);  
316         docObj = Sr_GetCurConvObjOfType(SR_OBJ_DOC);  
317         docId = Sr_GetDocId(docObj);  
318         elemId = Sr_GetFmElemId(srObj);  
319         ApplyCondition(docId, elemId, (StringT)"Form");  
320     }  
321     else if (F_StrIEqual(eventp->u.tag.gi,  
(StringT)"script"))  
322     {  
323         Sr_Convert(eventp, srObj);  
324         docObj = Sr_GetCurConvObjOfType(SR_OBJ_DOC);  
325         docId = Sr_GetDocId(docObj);  
326         elemId = Sr_GetFmElemId(srObj);  
327         ApplyCondition(docId, elemId, (StringT)"Script");  
328     }  
329  
330  
331     break;
```

```
332     case SR_EVT_RE:
333         break;
334
335     case SR_EVT_END_READER:
336         break;
337
338     default:
339         break;
340 }
341
342 return Sr_Convert(eventp, srObj);
343 }
```

Lines 1-8

These lines include header files for the import client. All import/export clients must include `fm_struct.h`, which defines the structure import/export API data structures and calls used by FrameMaker to import and export markup. `fm_struct.h` itself includes `fapi.h`, the FDK header file that defines standard FDK calls, and `fdetypes.h`, a header file that contains platform-independent primitive FDE data types used by all FDK calls, such as `BoolT` and `IntT`. For more information about replacing primitive data types, see Chapter 15, "Introduction to the FDE," in the *FDK Programmer's Guide*.

The client also uses supplemental FDE libraries that provide string manipulation, and platform-independent memory management functions that are declared in the separately-listed header files in this code. Using these the calls in these libraries enables you to write more portable code. For more information about using the FDE to write portable clients, see Chapter 15, "Introduction to the FDE," in the *FDK Programmer's Guide*.

In your own clients, you may need to include additional FDE and standard C header files at this point in your code. You may also want to define constants here.

Lines 10-22

These lines contain the function prototypes for the routines called during conversion object modification by `Sr_EventHandler()`. These functions are described above.

Depending on the platform you compile on, you may be required to provide function prototypes for the functions you create. In any case, ANSI C supports and recommends function prototypes as a way to ensure stricter type checking during compilation.

Lines 24-54

These lines contain static variable declarations and defined values. The definitions for `MetricT` values are standard technique for treating these units of measure as points. The table, row, and cell counters will be used to manage table importing issues such as nested tables in XHTML and determining the number of columns in a table.

Depending on the platform you compile on, you may be required to provide function prototypes for the functions you create. In any case, ANSI C supports and recommends function prototypes as a way to ensure stricter type checking during compilation.

Lines 56-343

These lines contain the code for the customized import event handler `Sr_EventHandler()`. All import/export clients that handle import must provide a customized event handler called `Sr_EventHandler()`. The import event handler for the XHTML client illustrates key features common to all import event handlers. The XHTML event handler performs the following tasks:

- Declares local variables for use throughout the routine.
- Uses a `switch` statement to examine all events passed to it by `FrameMaker`.
- Reacts to a subset of import events by modifying the associated conversion objects.
- Passes all event/conversion object pairs, whether modified or not, to the `FrameMaker` import conversion routine, `Sr_Convert()`.

Lines 72-78

In these lines the XHTML import event handler responds to the `SR_EVT_BEGIN_READER`, `SR_EVT_BEGIN_BOOK`, `SR_EVT_BEGIN_BOOK_COMP`, and `SR_EVT_BEGIN_DOC` events. These events occur once for each markup document to be imported. An event handler can respond to these events by performing any pre-processing initialization it needs to do—for example, writing a message to a log file. The XHTML event handler responds by getting the document ID for the current document object.

Lines 80-81

In these lines the XHTML import event handler simply passes any CDATA event out of the `switch` statement. While you could let this event fall through to the `switch` default, you may find that you want to trap this case in a future version of this import handler.

Lines 83-205

These lines handle events for the beginning of each element. The majority of this handling is devoted to tables—determining table dimensions, placing table heading and footing rows, and handling nested tables. The code also traps this event to handle instances of the `<a>` element.

The handler uses the following variables to track information about the current table elements as they're imported:

This variable:	Tracks the following:
<code>tableCount</code>	Tracks the level of table nesting. The value is incremented for the beginning of each <code>table</code> element, and decremented for the ending of each table element. If the value is greater than 1, the event is for a table within a nested table

This variable:	Tracks the following:
<code>tableRowCount</code>	Tracks the row number for the table. The value is incremented and decremented for beginning and ending events for <code>tr</code> elements, respectively. This is used to get the column width from the cells of the first table row.
<code>TD</code>	A boolean to track whether <code>FrameMaker</code> is in the process of importing content for a <code>td</code> element. This is important for handling nested tables—for example, if the event is for a table heading, but <code>TD</code> is <code>TRUE</code> , then the handler unwraps the heading and imports it as content of the current <code>FrameMaker</code> table cell.
<code>TH</code>	A boolean to track whether <code>FrameMaker</code> is in the process of importing content for a <code>th</code> element. Use is similar to the use of <code>TD</code> , above.

The handler traps begin element events and processes them as follows. In each case, if the handler doesn't specifically return, then the processing falls through to the call to `Sr_Convert()` at the end of the handler code block:

If the element GI is:	The event handler does the following:
<code>table</code>	Increments the <code>tableCount</code> variable. If the value of <code>tableCount</code> is greater than one, returns without processing the element event.
<code>tr</code>	If the value of <code>tableCount</code> equals one, increments the value of <code>tableRowCount</code> . If <code>tableCount</code> is greater than one, returns without processing the element.
<code>td</code>	If <code>tableCount</code> equals one, saves the insertion location and sets <code>TD</code> to <code>TRUE</code> . Then, if <code>tableRowCount</code> equals one, stores the cell's width attribute (if there is one) in the <code>tableWidths</code> array. This array will be used later by the <code>SetWidthOnImport()</code> function. If <code>tableCount</code> is greater than one, this is a cell in a nested table. The handler sets the insertion location to the last saved location, and specifies the <code>FrameMaker</code> element tag to use (if necessary). It then unwraps the XHTML element before converting it.
<code>th</code>	The processing is effectively the same as for <code>td</code> .
<code>tfoot</code>	If <code>tableCount</code> is greater than one, the handler unwraps the element so its content can be processed. Then it returns without processing the given element.
<code>thead</code>	
<code>caption</code>	
<code>tbody</code>	

If the element GI is:	The event handler does the following:
a	<p>First the handler converts the element. This is necessary because there is no corresponding FrameMaker element until after the conversion of the event. After the conversion, the handler gets the object ID for the FrameMaker element, and passes it to the <code>InsertMarker()</code> function. This function generates data necessary to recreate the link behavior in a FrameMaker document.</p> <p>Notice that the handler returns after processing this event. If the handler allowed the event to fall through, the event would be converted twice.</p>

Lines 210-331

These lines handle events for the ending of each element. The handler traps end element events and processes them as follows. In each case, if the handler doesn't specifically return, then the processing falls through to the call to `Sr_Convert()` at the end of the handler code block:

If the element GI is:	The event handler does the following:
table	<p>If <code>tableCount</code> equals one, the handler gets the document ID and the element ID for the current table. It passes this information to the <code>DeleteCols()</code> function, which deletes all empty columns from the table. It then processes the conversion event, and calls <code>SetWidthOnImport()</code>, <code>CenterOnImport()</code>, and <code>UpdateTblBorders()</code> to finish formatting the table. After this, it decrements <code>tableCount</code>. If the resulting value for <code>tableCount</code> is zero, it initializes the table tracking variables.</p> <p>If the value of <code>tableCount</code> is greater than one, the handler decrements the value of <code>tableCount</code>, unwraps the element, and converts the event.</p>
tr	<p>If the value of <code>tableCount</code> equals one, updates the count of total cells (if necessary), and resets <code>cellCount</code> to zero. If <code>tableCount</code> is greater than one, unwraps the row element before processing the event.</p>
td	<p>If <code>tableCount</code> equals one, converts the event, saves the insertion location and increments <code>cellCount</code>. Then it returns, to avoid converting the event again. If <code>tableCount</code> is greater than one, the handler unwraps the element before converting the event.</p>
th	<p>The processing is effectively the same as for <code>td</code>.</p>

If the element GI is:	The event handler does the following:
tfoot	If <code>tableCount</code> is greater than one, the handler unwraps the element so its content can be processed. Then it returns without processing the given element.
thead	
caption	
tbody	
label	These are all forms elements. The handler first converts the event, then passing the resulting element ID to <code>ApplyCondition()</code> so the user can hide these elements.
select	
optgroup	
option	
textarea	
fieldset	
button	
isindex	
input	
script	
	Processed the same as the forms elements above.

Lines 332-343

These lines finish the event handler code block. The remaing switch cases fall through to line 342, which converts the event.

XHTML export example

For the export of XHTML the starter kit defines seven functions:

- `GetTableWidth()` to get the width of the FrameMaker table before exporting
- `GetTableAlignment()` to determine the value for the table element's alignment attribute
- `GetCellWidth()` to get the widths of columns
- `GetBorderFromRuling()` to translate FrameMaker table ruling into border attributes
- `GetTableRuling()` to get the table ruling values that are used in `GetBorderFromRuling()`
- `GetTableFrame()` to convert table ruling properties into table frame attributes

These functions are implemented in the import module, but this example does not show their implementation. You can see these functions in the source code that is included in the XHTML starter kit.

```
1  #include "fm_sgml.h"
2  #include "fmemory.h"
3  #include "fstrings.h"
4  #include "fstrlist.h"
5  #include "fmetrics.h"
6  #include "futils.h"
7
8  #define pts (MetricT)65536
9
10 StringT GetTableWidth(F_ObjHandleT docId, F_ObjHandleT
tableId);
11 StringT GetTableAlignment(F_ObjHandleT docId, F_ObjHandleT
tableId);
12 StringT GetCellWidth(F_ObjHandleT docId, F_ObjHandleT cellId);
13 StringT GetBorderFromRuling(F_ObjHandleT docId, F_ObjHandleT
fmtRulingId);
14 StringT GetTableBorder(F_ObjHandleT docId, F_ObjHandleT
tableId);
15 StringT GetTableRuling(F_ObjHandleT docId, F_ObjHandleT tblId);
16 StringT GetTableFrame(F_ObjHandleT docId, F_ObjHandleT tblId);
17
18 static SrwTablePartTypeT XhtmlScanOrder[4] = {SRW_TABLE_TITLE,
19                                               SRW_TABLE_HEADING,
20                                               SRW_TABLE_FOOTING,
21                                               SRW_TABLE_BODY};
22 /*
23  * XML Writer Sample Event Handler for XHTML
24  */
25 SrwErrorT Sw_EventHandler(eventp, swObj)
26 SwEventT *eventp;
27 SwConvObjT swObj;
28 {
29     SgmlAttrValT att;
30     SwConvObjT docObj;
31     static F_ObjHandleT docId;
32     StringT width = NULL, align = NULL, border = NULL,
33           rules = NULL, frame = NULL;
34
35     switch (Sw_GetObjType(swObj))
36     {
37         case SW_OBJ_TABLE:
```

```
38      docObj = Sw_GetCurConvObjOfType(SW_OBJ_DOC);
39      if (!docObj)
40          docObj = Sw_GetCurConvObjOfType(SW_OBJ_BOOK_COMP);
41
42      if (docObj)
43      {
44          docId = Sw_GetDocId(docObj);
45
46          /* set table width in points. */
47          width = GetTableWidth(docId, eventp->fm_objid);
48          att = Sw_GetAttrVal(swObj, (StringT)"width");
49          att.sgmlAttrVal = (StringListT) F_StrListNew(1,1);
50          F_StrListAppend(att.sgmlAttrVal, width);
51          Sw_SetAttrVal(swObj, &att);
52
53          Structured_DeallocateAttrVal(&att);
54          F_ApiDeallocateString(&width);
55
56
57          /* set table align attribute */
58          align = GetTableAlignment(docId, eventp->fm_objid);
59
60          att = Sw_GetAttrVal(swObj, (StringT)"align");
61          att.sgmlAttrVal = (StringListT) F_StrListNew(1,1);
62          F_StrListAppend(att.sgmlAttrVal, align);
63          Sw_SetAttrVal(swObj, &att);
64
65          Structured_DeallocateAttrVal(&att);
66          F_ApiDeallocateString(&align);
67
68          /* set table borders */
69          border = GetTableBorder(docId, eventp->fm_objid);
70          att = Sw_GetAttrVal(swObj, (StringT)"border");
71          att.sgmlAttrVal = (StringListT) F_StrListNew(1,1);
72          F_StrListAppend(att.sgmlAttrVal, border);
73          Sw_SetAttrVal(swObj, &att);
74
75          Structured_DeallocateAttrVal(&att);
76          F_ApiDeallocateString(&border);
77
78          /* set table ruling */
79          rules = GetTableRuling(docId, eventp->fm_objid);
80          att = Sw_GetAttrVal(swObj, (StringT)"rules");
81          att.sgmlAttrVal = (StringListT) F_StrListNew(1,1);
```

```
82         F_StrListAppend(att.sgmlAttrVal, rules);
83         Sw_SetAttrVal(swObj, &att);
84
85         Structured_DeallocateAttrVal(&att);
86         F_ApiDeallocateString(&rules);
87
88         /* set table frame */
89         /* note this overwrites the attr value set
90          * by FM since it tries to write a CALS table
91          * instead of an XHTML table.
92          */
93         frame = GetTableFrame(docId, eventp->fm_objid);
94         att = Sw_GetAttrVal(swObj, (StringT)"frame");
95         att.sgmlAttrVal = (StringListT) F_StrListNew(1,1);
96         F_StrListAppend(att.sgmlAttrVal, frame);
97         Sw_SetAttrVal(swObj, &att);
98
99         Structured_DeallocateAttrVal(&att);
100        F_ApiDeallocateString(&frame);
101
102        /* Swap tbody and tfoot order */
103        Sw_SetTableScanOrder(swObj, XhtmlScanOrder);
104    }
105    break;
106
107    case SW_OBJ_TABLE_CELL:
108        docObj = Sw_GetCurConvObjOfType(SW_OBJ_DOC);
109        if (!docObj)
110            docObj = Sw_GetCurConvObjOfType(SW_OBJ_BOOK_COMP);
111
112        if (docObj)
113        {
114            docId = Sw_GetDocId(docObj);
115
116            width = GetCellWidth(docId, eventp->fm_objid);
117            att = Sw_GetAttrVal(swObj, (StringT)"width");
118            att.sgmlAttrVal = (StringListT) F_StrListNew(1,1);
119            F_StrListAppend(att.sgmlAttrVal, width);
120            Sw_SetAttrVal(swObj, &att);
121
122            Structured_DeallocateAttrVal(&att);
123            F_ApiDeallocateString(&width);
124        }
125    break;
```

```
126
127     default:
128         break;
129
130     }
131
132     return Sw_Convert(eventp, swObj);
133 }
```

Lines 1-21

Similar to the beginning lines for the import event handler, these lines include header files, declare function prototypes, and define static variables.

Lines 25-133

These lines contain the code for the customized export event handler `Sw_EventHandler()`. All import/export clients that handle export must provide a customized event handler called `Sw_EventHandler()`.

Aside from handling export events, the export event handler for the XHTML client differs from the import handler in the criteria for the switch statement. Instead of trapping events, this handler traps the type of object that triggers the event. This is because the export handler only needs to trap FrameMaker tables and table cells.

The XHTML event handler performs the following tasks:

- Declares local variables for use throughout the routine.
- Uses a `switch` statement to examine all events passed to it by FrameMaker.
- Reacts to a subset of import events by modifying the associated conversion objects.
- Passes all event/conversion object pairs, whether modified or not, to the FrameMaker import conversion routine, `Sr_Convert()`.

Lines 35-105

In these lines the XHTML export event handler responds to each FrameMaker table in the document being saved as XML. The first thing it does is get the ID of the current document (see lines 38-44). This is important because the handler uses standard FDK calls to inspect the FrameMaker document—Nearly all FDK calls require a document ID. For example, the function `GetTableWidth()` uses FDK calls to get data about the table as it is in the FrameMaker document.

The remainder of this block of code gets values from the FrameMaker table, and writes them as attribute values in the proposed conversion event. After writing the values, the handler lets the event fall through the switch statement to the call to `Sw_Convert()`.

Lines 47-54 illustrate this process. The handler first uses `GetTableWidth` to get a string that represents the table width as measured in points. It then gets the proposed width attribute

for this conversion object, and adds this value to it. It finally sets this new attribute structure to the conversion object, and frees all the allocated memory.

The remaining code in this block uses the same technique to set other table attributes.

After setting all the attributes, the handler uses a function called `Sw_SetTableScanOrder()`. This function specifies ultimate the order of these table components in the target file. It's a necessary function because FrameMaker is not flexible in the order of these table components, yet SGML and XML are.

`Sw_SetTableScanOrder()` receives an array of integers that represent the basic table components—table title, table heading, table body, and table footing. The array is declared on line 18 of the code. The order of the integers in this array determine the order of the elements in the target file. For XHTML the order is set to table title, table heading, table footing, table body.

Lines 107-125

In these lines the XHTML export event handler sets the cell's width attribute in the same way as it sets attributes for the table.

Lines 132-133

These lines finish the event handler code block by calling `Sw_Convert()` to convert the event.

Conclusion

This code shows examples of various techniques you can use when developing an import/export client. Some of these techniques include:

- Case statements to trap conversion events either by event or by object type
- Unwrapping elements upon conversion
- Using the `Sw_SetTableScanOrder()` function to change the order of table elements upon conversion
- Returning without converting the event
- Getting and setting data structures for the proposed conversion object
- Using the FDK to inspect or modify the FrameMaker document

By no means does this example exhaust the possible techniques for import/export clients. For example, you can use the Structure Import/Export API to:

- Change the order of any elements (not just table components)
- Write markup directly in the target file on export
- Manipulate values for ID and IDREF attributes
- Many other useful actions

4

Structure Import/Export API Function Reference

This chapter is an alphabetical list of the structure import/export functions used to write import/export API clients. These functions fall into four broad classes as outlined in the following table:

Class of functions	Purpose
Functions prefixed by <code>Sgml_</code>	Deprecated functions. With FrameMaker 7.0 these functions have been renamed to use a prefix of <code>Structured_</code> . For backward compatibility all <code>Sgml_</code> functions names are defined to equal their counterpart <code>Structured_</code> functions.
Functions prefixed by <code>Structured_</code> , <code>EndTagOmissible()</code> , and <code>StartTagOmissible()</code>	Read values from the XML or SGML DTD, or from the SGML declaration.
Functions prefixed by <code>Sr_</code>	Retrieve, set, or convert values associated with import conversion objects.
Functions prefixed by <code>Srw_</code>	Retrieve, set, or delete values associated with import or export conversion objects.
Functions prefixed by <code>Sw_</code>	Retrieve, set, or convert values associated with export conversion objects.

Some examples in this chapter use standard FDK or FDE functions in addition to using structure import/export API functions. For more information on using the FDK and FDE functions, see the *FDK Programmer's Reference* and the *FDK Programmer's Guide*.

Important: The examples in this chapter assume that your client includes the `fm_struct.h` header file.

EndTagOmissible()

Tests whether an element's end-tag can be omitted in a document instance. Note that end-tag omission is not valid for XML.

Synopsis

```
#include "fm_struct.h"

. . .

IntT EndTagOmissible (StructuredElementDefT *elemDef);
```

Arguments

elemDef Pointer to the element definition to query

Details

`EndTagOmissible()` is a macro that tests the `tagOmission` field of *elemDef* to determine if `OMIT_END_TAG` is set. If set, it indicates that the end-tag can be omitted. This macro can be used only on element definitions, not element instances.

Returns

`OMIT_END_TAG` if the end-tag is omissible; otherwise 0. For XML always returns 0.

Examples

The following code shows an import event handler; for the begin element event, the code gets the element definition, then checks to see whether end tag can be omitted.

```
. . .
SrErrorT Sr_EventHandler(eventp, srObj)
    SrEventT *eventp;
    SrConvObjT srObj;
{
    StructuredElementDefT *elemDefp;

    switch(eventp->evtype) {
        case SR_EVT_BEGIN_ELEM:
            elemDefp = Structure_GetElementDef(eventp->u.tag.gi);
            if (EndTagOmissible (elemDefp))
                /* end tag can be omitted, so process accordingly. */
                {
                    . . .
                }
    }
}
```

See also

- [“StartTagOmissible\(\)” on page 278](#)
- [“StructuredElementDefT” on page 413](#)
- [“Primitive data types” on page 383](#) for more information on `BoolT`

Structured_CopyAttrVal()

Deprecated: *Sgml_CopyAttrVal()*

Copies a single attribute from markup.

Synopsis

```
#include "fm_struct.h"

. . .
StructuredAttrValT Structure_CopyAttrVal(StructuredAttrValT
*attrVal);
```

Arguments

attrVal Pointer to the markup attribute to copy

Details

When you no longer need the copy of the attribute value, free the memory allocated for it by calling `Structured_DeallocateAttrVal()`.

Returns

An attribute structure that duplicates the structure pointed to by *attrVal*, or a zeroed out structure on error.

Examples

The following code finds the element named `article` and copies its `version` attribute value to a global variable. It then sets that value to every other child of `article` that includes a `version` attribute. Note where the code deallocates memory for the local and global attribute values. The code assumes `article` is the highest level element in the document; since it deallocates the global attribute value in the end element event for `article`, any parent of `article` would trigger an invalid call to `Sw_SetAttrVal()`.

```
. . .
static StructuredAttrValT staticAttrVal;

SrwErrorT Sw_EventHandler(eventp, swObj)
    SwEventT *eventp;
    SwConvObjT swObj;
{
    StructuredAttrValT tempAttrVal;
```

```
switch(eventp->evtype) {
case SW_EVT_BEGIN_ELEM:
    if(F_StrIEqual(Sw_GetStructuredGi(swObj),
                    (StringT)"Article")) {
        tempAttrVal = Sw_GetAttrVal(swObj, (StringT)"version");
        staticAttrVal = Structured_CopyAttrVal(&tempAttrVal);
    } else {
        tempAttrVal = Sw_GetAttrVal(swObj, (StringT)"version");
        if(!F_StrIsEmpty(tempAttrVal.sgmlAttrName)) {
            Sw_SetAttrVal(swObj, &staticAttrVal);
        }
    }
    Structured_DeallocateAttrVal(&tempAttrVal);
    break;
case SW_EVT_END_ELEM:
    if(F_StrIEqual(Sw_GetStructuredGi(swObj),
                    (StringT)"Article"))
        Structured_DeallocateAttrVal(&staticAttrVal);
    break;
default:
    break;
}
. . .
```

See also

- [“Structured_DeallocateAttrVal\(\)” on page 55](#)
- [“StructuredAttrValT” on page 412](#)

Structured_CopyAttrVals()

Deprecated: *Sgml_CopyAttrVals()*

Copies a list of attributes from markup.

Synopsis

```
#include "fm_struct.h"

. . .
StructuredAttrValsT Structured_CopyAttrVals(StructuredAttrValsT
*attrVals);
```

Arguments

attrVals Pointer to the attribute list to copy

Details

When you no longer need the copy of the attribute value list, free the memory allocated for it by calling `Structured_DeallocateAttrVals()`.

Returns

An attribute list structure that duplicates the structure pointed to by *attrVals*, or a zeroed out structure on error.

Examples

The following code copies the attribute list from the first occurrence of an `article` element, and then sets that attribute list to every occurrence of `report` elements. Note that the code deallocates the attribute values list.

```
. . .
static StructuredAttrValsT staticAttrVals;
BoolT haveArticle = False;

SrwErrorT Sw_EventHandler(eventp, swObj)
    SwEventT *eventp;
    SwConvObjT swObj;
{
    StructuredAttrValsT tempAttrVals;
    switch(eventp->evtype) {
    case SW_EVT_BEGIN_ELEM:
        if(F_StrIEqual(Sw_GetStructuredGi(swObj),
(StringT)"Article")
                                && haveArticle == False) {
            tempAttrVals = Sw_GetAttrVals(swObj);
            staticAttrVals = Structured_CopyAttrVals(&tempAttrVals);
            Structured_DeallocateAttrVals(&tempAttrVals);
            haveArticle = True;
```

```
        } else if( F_StrIEqual(
                    Sw_GetStructuredGi(swObj), (StringT)"Report"))
            Sw_SetAttrVal(swObj, &staticAttrVal);
        break;
        /* You can deallocate here, or for the end event for */
        /* the parent of all report and article elements. */
        case SW_EVT_END_WRITER:
            if(F_StrIEqual(Sw_GetStructuredGi(swObj),
                (StringT)"Article"))
                Structured_DeallocateAttrVals(&staticAttrVals);
            break;
        default:
            break;
    }
    . . .
```

See also

- [“Structured_DeallocateAttrVal\(\)” on page 55](#)
- [“StructuredAttrValT” on page 412](#)
- [“StructuredAttrValsT” on page 413](#)

Structured_DeallocateAttrVal()

Deprecated: `Sgml_DeallocateAttrVal()`

Frees memory allocated for a specified attribute.

Synopsis

```
#include "fm_struct.h"

. . .

VoidT Structured_DeallocateAttrVal(StructuredAttrValT *attrVal);
```

Arguments

attrVal Pointer to the attribute to free

Details

This function performs a deep deallocation, freeing the structure, the string list, and all strings that were allocated for it. To free a list of attribute values, call `Structured_DeallocateAttrVals()`.

Returns

`VoidT`.

Examples

See the example for [“Structured_CopyAttrVal\(\)” on page 51](#).

See also

- [“Structured_DeallocateAttrVals\(\)” on page 56](#)
- [“StructuredAttrValT” on page 412](#)

Structured_DeallocateAttrVals()

Deprecated: Sgml_DeallocateAttrVals()

Frees memory allocated for a specified list of attribute values.

Synopsis

```
#include "fm_struct.h"

. . .
VoidT Structured_DeallocateAttrVals(StructuredAttrValsT
*attrVals);
```

Arguments

attrVals Pointer to the StructuredAttrValsT structure to free

Details

StructuredAttrValsT is a list of StructuredAttrValT structures. This function performs a deep deallocation, freeing the structure, and the list of StructuredAttrValT structures that were allocated for it. To free a single attribute value and free the memory allocated for it, call Structured_DeallocateAttrVal().

Returns

VoidT.

Examples

See the example for [“Structured_CopyAttrVals\(\)” on page 53](#).

See also

- [“Structured_DeallocateAttrVal\(\)” on page 55](#)
- [“StructuredAttrValsT” on page 413](#)

Structured_GetAppinfo()

Deprecated: Sgml_GetAppinfo()

Gets the *APPINFO* parameter from the declaration for the current markup document.

Synopsis

```
#include "fm_struct.h"
. . .
ConStringT Structured_GetAppinfo(VoidT);
```

Arguments

None.

Details

To retrieve the text string describing application-specific information from the SGML declaration, call `Structured_GetAppinfo()`.

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

When you import a document, FrameMaker looks for the SGML declaration in the following order and locations:

1. An explicit declaration preceding the DTD in the document you import
2. A specification in a subset of your application
3. The default declaration provided as part of FrameMaker

When you export a document, an SGML declaration can be provided as a subset of your application or can be part of the export DTD.

Note that the SGML declaration is specific to the markup document instance. This means you cannot reliably get the *APPINFO* parameter until after a begin event for either a book, book component, or document. This is true for both importing and exporting.

Returns

The *APPINFO* text string, or `NULL` if the current application is `NONE`. For XML, always returns `NULL`.

Examples

The following code retrieves the *APPINFO* parameter from the declaration for the current markup document:

```
. . .
ConStringT appInfo;
. . .
appInfo = Structured_GetAppinfo();
. . .
```

See also

- [“Structured_GetDocTypeName\(\)” on page 61](#)
- [“Structured_GetLoc\(\)” on page 75](#)
- [“Primitive data types” on page 383](#) for more information on ConStringT

Structured_GetDefaultEntityDef()

Deprecated: Sgml_GetDefaultEntityDef()

Retrieves the definition of the default entity for the current SGML document.

Synopsis

```
#include "fm_struct.h"
. . .
const StructuredEntityDefT
*Structured_GetDefaultEntityDef(VoidT);
```

Arguments

None.

Details

When a default entity is declared, call `Structured_GetDefaultEntityDef()` to retrieve a pointer to a structure containing a description of the default entity.

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

Note that the DTD is specific to the document instance. This means you cannot reliably get the default entity definition until after a begin event for either a book, book component, or document. This is true for both importing and exporting.

Returns

A pointer to a `StructuredEntityDefT` structure, or `NULL` if a default entity is not declared. For XML always returns `NULL`.

Examples

The following code returns a pointer to a structure describing the default entity:

```
. . .
const StructuredEntityDefT *defEntity;
. . .
defEntity = Structured_GetDefaultEntityDef();
. . .
```

See also

- [“Structured_GetEntityDef\(\)” on page 63](#)
- [“Structured_GetFirstEntityName\(\)” on page 68](#)
- [“Structured_GetNextEntityName\(\)” on page 78](#)
- [“StructuredEntityDefT” on page 414](#)

Structured_GetDelimiterString()

Deprecated: Sgml_GetDelimiterString()

Returns the character string used as a specified delimiter.

Synopsis

```
#include "fm_struct.h"
. . .
ConStringT Structured_GetDelimiterString(StructuredDelimiterTypeT
deltype);
```

Arguments

deltype The delimiter type for which to retrieve a character string. For a list of possible values, see [“StructuredDelimiterTypeT” on page 387](#).

Details

To examine the character string specified as a particular delimiter in an SGML declaration, call `Structured_GetDelimiterString()`. The parameter *deltype* is one of the enumerated delimiter types specified in `StructuredDelimiterTypeT`.

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

When you import a document, FrameMaker looks for the SGML declaration in the following order and locations:

1. An explicit declaration preceding the DTD in the document you import

2.A specification in a subset of your application

3.The default declaration provided as part of FrameMaker

When you export a document, an SGML declaration can be provided as a subset of your application or can be part of the export DTD.

Note that the SGML declaration is specific to the document instance. This means you cannot reliably get the delimiter string until after a begin event for either a book, book component, or document. This is true for both importing and exporting.

For XML the SGML declaration is implied. This means that you cannot specify an SGML declaration, and the delimiter strings will not vary from the XML specification. However, if you use this function to get a delimiter string for XML, you should wait for the begin event for the book, book component, or document.

Returns

The string used as the specified delimiter.

Examples

The following code retrieves the delimiter string for the STRUCTURED_DE_RNI delimiter. STRUCTURED_DE_RNI corresponds to the reserved name indicator (*RNI*) delimiter:

```
. . .  
ConStringT delimStr;  
. . .  
delimStr = Structured_GetDelimiterString(STRUCTURED_DE_RNI);  
. . .
```

See also

- [“Structured_GetAppinfo\(\)” on page 57](#)
- [“Structured_GetEntityNamecase\(\)” on page 65](#)
- [“Structured_GetGeneralNamecase\(\)” on page 71](#)
- [“Structured_GetLcnmchar\(\)” on page 73](#)
- [“Structured_GetLcnmstr\(\)” on page 74](#)
- [“Structured_GetQuantity\(\)” on page 85](#)
- [“Structured_GetReservedName\(\)” on page 87](#)
- [“Structured_GetUcnmchar\(\)” on page 89](#)
- [“Structured_GetUcnmstr\(\)” on page 90](#)
- [“StructuredDelimiterTypeT” on page 387](#)
- [“Primitive data types” on page 383](#) for more information on ConStringT

Structured_GetDocTypeName()

Deprecated: `Sgml_GetDocTypeName()`

Retrieves the document type name for the current markup document from the DTD.

Synopsis

```
#include "fm_struct.h"
. . .
ConStringT Structured_GetDocTypeName(VoidT);
```

Arguments

None.

Details

To retrieve the document type name for the current document from the DTD, call `Structured_GetDocTypeName()`. The document type name is the generic identifier (*GI*) of the document element.

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

On import, the DTD is specified as part of the document instance. On export, the DTD is specified within the current structure application. Note that the DTD is specific to the document instance. This means you cannot reliably get the document type name until after a begin event for either a book, book component, or document. This is true for both importing and exporting.

Returns

A string containing the name of the document type.

Examples

The following code returns the document type name of the current markup document:

```
. . .
ConStringT docType;
. . .
docType = Structured_GetDocTypeName();
. . .
```

See also

- [“Structured_GetAppinfo\(\)” on page 57](#)
- [“Primitive data types” on page 383](#) for more information on `ConStringT`

Structured_GetElementDef()

Deprecated: `Sgml_GetElementDef()`

Returns the definition of the specified element in the DTD.

Synopsis

```
#include "fm_struct.h"

. . .
const StructuredElementDefT *Structured_GetElementDef(StringT
etag);
```

Arguments

etag Name of the element to query

Details

To retrieve a pointer to the structure containing the definition of a specified element in a DTD, pass the name of the element to `Structured_GetElementDef()`.

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

For SGML the reference concrete syntax is case-insensitive for element names, so *etag* can be passed in lowercase, uppercase, or mixed case. For XML, or if the SGML declaration uses naming rules to override the SGML reference concrete syntax, *etag* is case-sensitive.

On import, the DTD is specified as part of the document instance. On export, the DTD is specified within the current structure application.

Returns

A pointer to an `StructuredElementDefT` structure, or `NULL` if there is no declaration for this element.

Examples

The following code shows a fragment of an import event handler that loops through all the element definitions in the DTD and returns the definition structure. Note that since the DTD is associated with a given document instance, the handler waits for a begin document event before processing the element definitions.

```
. . .
const StructuredElementDefT *elemDefp;
ConStringT elemName;
. . .
```

```
switch(eventp->evtype) {
case SR_EVT_BEGIN_DOC:
    . . .
    elemName = Structured_GetFirstElementName();
    while (elemName) {
        elemDefp = Structured_GetElementDef((StringT)elemName);
        /* Process the element definition as desired. */
        elemName = Structured_GetNextElementName((StringT)
elemName);
    }
    . . .
```

See also

- [“Structured_GetFirstElementName\(\)” on page 67](#)
- [“Structured_GetNextElementName\(\)” on page 77](#)
- [“StructuredElementDefT” on page 413](#)
- [“Primitive data types” on page 383](#) for more information on StringT

Structured_GetEntityDef()

Deprecated: Sgml_GetEntityDef()

Retrieves the definition of a specified entity.

Synopsis

```
#include "fm_struct.h"
. . .
const StructuredEntityDefT
*Structured_GetEntityDef(StructuredEntityScopeT scope,
StringT ename);
```

Arguments

<i>scope</i>	STRUCTURED_ES_GENERAL for a general entity, or STRUCTURED_ES_PARAMETER for a parameter entity
<i>ename</i>	Name of the entity to query

Details

This function can get the definition of a general entities or a parameter entities; general entities can be defined in either the external DTD or in the internal document subset. The *scope* parameter indicates the type of entity to query. *ename* is the name of the entity to query.

When getting the definition of a parameter entity, be sure *ename* does not include the parameter reference open character (*PERO*).

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

Note that this function is most useful within the entity handler, `Srw_EntityHandler()`. For more information, see [“Entity handlers” on page 25](#) and [“Srw_EntityHandler\(\)” on page 259](#).

For SGML the reference concrete syntax is case-insensitive for element names, so *ename* can be passed in lowercase, uppercase, or mixed case. For XML, or if the SGML declaration uses naming rules to override the SGML reference concrete syntax, *ename* is case-sensitive.

On import, the DTD is specified as part of the document instance. On export, the DTD is specified within the current structure application.

Returns

A pointer to a `StructuredEntityDefT` structure that contains the entity definition, or `NULL` if the specified entity is not found.

Examples

The following code shows an entity handler that prints out entity definition data for every entity reference event.

```
. . .
FilePathT *Srw_EntityHandler(
    StringT entName,
    StructuredEntityScopeT scope,
    FilePathT *defaultFp)
{
    const StructuredEntityDefT *entDef;

    entDef = Structured_GetEntityDef(scope, entName);
    if(!F_StrIsEmpty(entDef->ename))
        F_Printf(NULL, "\n    ename: %s", entDef->ename);
    if(!F_StrIsEmpty(entDef->etext))
        F_Printf(NULL, "\n    etext: %s", entDef->etext);
    if(!F_StrIsEmpty(entDef->pubid))
        F_Printf(NULL, "\n    pubid: %s", entDef->pubid);
    if(!F_StrIsEmpty(entDef->sysid))
        F_Printf(NULL, "\n    sysid: %s", entDef->sysid);
    if(!F_StrIsEmpty(entDef->nname))
        F_Printf(NULL, "\n    nname: %s", entDef->nname);
}
```



```
    if(entDef->external)
        F_Printf(NULL, "\n        external: TRUE");
    else
        F_Printf(NULL, "\n        external: FALSE");

    /* Return the default file path so entity is */
    /* not changed by the handler. */
    return(defaultFp);
}
. . .
```

See also

- [“Structured_GetEntityNamecase\(\)” on page 65](#)
- [“Structured_GetFirstEntityName\(\)” on page 68](#)
- [“Structured_GetNextEntityName\(\)” on page 78](#)
- [“Srw_EntityHandler\(\)” on page 259](#)
- [“StructuredEntityDefT” on page 414](#)
- [“StructuredEntityScopeT” on page 388](#)

Structured_GetEntityNamecase()

Deprecated: Sgml_GetEntityNamecase()

Returns the status of the NAMECASE ENTITY parameter in the SGML declaration.

Synopsis

```
#include "fm_struct.h"
. . .
BoolT Structured_GetEntityNamecase(VoidT);
```

Arguments

None.

Details

To determine the case sensitivity of entity names according to the *NAMECASE ENTITY* parameter in the SGML declaration for a document, call `Structured_GetEntityNamecase()`.

If *NAMECASE ENTITY* is *NO*, this function returns *False*, meaning that case is significant for entity names. If no SGML declaration is specified for an SGML application, *FrameMaker* assumes a value of *NO*.

If *NAMECASE ENTITY* is *YES*, then `Structured_GetEntityNamecase()` returns *True*, meaning that case is not significant for entity names.

When you import a document, FrameMaker looks for the SGML declaration in the following order and locations:

1. An explicit declaration preceding the DTD in the document you import
2. A specification in a subset of your application
3. The default declaration provided as part of FrameMaker

When you export a document, an SGML declaration can be provided as a subset of your application or can be part of the export DTD.

Note that the SGML declaration is specific to the document instance. This means you cannot reliably get the NAMECASE ENTITY parameter until after a begin event for either a book, book component, or document. This is true for both importing and exporting.

Returns

True or False. For XML always returns False.

Examples

The following code retrieves the status of *NAMECASE ENTITY* from an SGML declaration:

```
. . .  
BoolT entityIsCapped;  
. . .  
entityIsCapped = Structured_GetEntityNamecase();  
. . .
```

See also

- [“Structured_GetGeneralNamecase\(\)” on page 71](#)
- [“Structured_GetEntityDef\(\)” on page 63](#)
- [“Structured_GetFirstEntityName\(\)” on page 68](#)
- [“Structured_GetNextEntityName\(\)” on page 78](#)
- [“Primitive data types” on page 383](#) for more information on BoolT

Structured_GetFirstElementName()

Deprecated: Sgml_GetFirstElementName()

Retrieves the generic identifier (GI) of the first element in the current DTD for which a declaration is reported by the parser.

Synopsis

```
#include "fm_struct.h"
. . .
ConStringT Structured_GetFirstElementName(VoidT);
```

Arguments

None.

Details

This function is useful to begin a loop through all the elements declared in a DTD; use it in conjunction with `Structured_GetNextElementName()`. You can pass the name returned by this function to `Structured_GetElementDef()` to retrieve a pointer to the structure containing the element's definition.

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

For SGML the reference concrete syntax is case-insensitive for element names. For XML, or if the SGML declaration uses naming rules to override the SGML reference concrete syntax, element names are case-sensitive.

On import, the DTD is specified as part of the document instance. On export, the DTD is specified within the current structure application.

Returns

The *GI* of the first defined element in the DTD.

Examples

The following code retrieves the names of all declared elements in a DTD:

```
. . .
const StructuredElementDefT *elemDefp;
ConStringT elemName;
. . .
```

```
switch(eventp->evtype) {
case SR_EVT_BEGIN_DOC:
. . .
    elemName = Structured_GetFirstElementName();
    while (elemName) {
        elemDefp = Structured_GetElementDef((StringT)elemName);
        /* Process the element definition as desired. */
        elemName = Structured_GetNextElementName((StringT)
                                                    elemName);
    }
. . .
```

See also

- [“Structured_GetNextElementName\(\)” on page 77](#)
- [“Structured_GetElementDef\(\)” on page 62](#)
- [“Structured_GetGeneralNamecase\(\)” on page 71](#)
- [“Primitive data types” on page 383](#) for more information on ConStringT and StringT

Structured_GetFirstEntityName()

Deprecated: Sgml_GetFirstEntityName()

Retrieves the name of the first general entity or parameter entity in the DTD that has a definition reported by the parser.

Synopsis

```
#include "fm_struct.h"
. . .
ConStringT Structured_GetFirstEntityName(StructuredEntityScopeT
scope);
```

Arguments

<i>scope</i>	STRUCTURED_ES_GENERAL for a general entity, or STRUCTURED_ES_PARAMETER for a parameter entity
--------------	--

Details

This function is useful to begin a loop through all the declared entities; use it in conjunction with `Structured_GetNextEntityName()`. You can pass the name returned by this

function to `Structured_GetEntityDef()` to retrieve a pointer to the structure containing the entity's definition.

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

scope indicates the type of entity to search for. If *scope* is set to `STRUCTURED_ES_PARAMETER`, the entity name returned by this function is preceded by its initial parameter entity reference open (*PERO*) delimiter.

For SGML the reference concrete syntax is case-insensitive for entity names. For XML, or if the SGML declaration uses naming rules to override the SGML reference concrete syntax, entity names are case-sensitive.

On import, the DTD is specified as part of the document instance. On export, the DTD is specified within the current structure application.

Returns

The name of the first entity in the DTD.

Examples

The following code retrieves the definitions of all declared parameter entities in a DTD:

```
. . .
ConStringT entName;
const StructuredEntityDefT *entDefp;
. . .
entName = Structured_GetFirstEntityName(STRUCTURED_ES_PARAMETER);
while (entName)
{
    entDefp = Structured_GetEntityDef(STRUCTURED_ES_PARAMETER,
                                     (StringT)entName);
    /* Now process the entity definition. */
    entName =
        Structured_GetNextEntityName(STRUCTURED_ES_PARAMETER,
                                     (StringT)entName);
}
. . .
```

See also

- [“Structured_GetNextEntityName\(\)” on page 78](#)
- [“Structured_GetEntityDef\(\)” on page 63](#)
- [“Structured_GetEntityNamecase\(\)” on page 65](#)
- [“StructuredEntityScopeT” on page 388](#)

- [“Primitive data types” on page 383](#) for more information on `ConStringT` and `StringT`

Structured_GetFirstNotationName()

Deprecated: `Sgml_GetFirstNotationName()`

Retrieves the name of the first data-content notation in the DTD that has a definition reported by the parser.

Synopsis

```
#include "fm_struct.h"

. . .

ConStringT Structured_GetFirstNotationName(VoidT);
```

Arguments

None.

Details

This function is useful to begin a loop through all the declared entities; use it in conjunction with `Structured_GetNextNotationName()`. You can pass the name returned by this function to `Structured_GetNotationDef()` to retrieve a pointer to the structure containing the notation's definition.

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

For SGML the reference concrete syntax is case-insensitive for notation names. For XML, or if the SGML declaration uses naming rules to override the SGML reference concrete syntax, notation names are case-sensitive.

On import, the DTD is specified as part of the document instance. On export, the DTD is specified within the current structure application.

Returns

The name of the first data-content notation in the DTD.

Examples

The following code retrieves the names of all declared data-content notations in a DTD:

```
. . .
ConStringT notateName;
const StructuredNotationDefT *notationp;
. . .
notateName = Structured_GetFirstNotationName();
```

```
while (notateName)
{
    notationp = Structured_GetNotationDef((StringT)notateName);
    /* Process the notation definition here */
    notateName = Structured_GetNextNotationName(
        (StringT)notateName);
}
. . .
```

See also

- [“Structured_GetNextNotationName\(\)” on page 80](#)
- [“Structured_GetNotationDef\(\)” on page 82](#)
- [“Structured_GetGeneralNamecase\(\)” on page 71](#)
- [“Primitive data types” on page 383](#) for more information on `ConStringT` and `StringT`

Structured_GetGeneralNamecase()

Deprecated: Sgml_GetGeneralNamecase()

Returns the status of the *NAMECASE GENERAL* parameter in the SGML declaration.

Synopsis

```
#include "fm_struct.h"
. . .
BoolT Structured_GetGeneralNamecase(VoidT);
```

Arguments

None.

Details

To determine the case sensitivity of attribute, element, and notation names according to the *NAMECASE GENERAL* parameter in the SGML declaration for a document, call `Structured_GetGeneralNamecase()`. To determine the case-sensitivity of entity names, call `Structured_GetEntityNamecase()` instead.

If *NAMECASE GENERAL* is YES (this is the FrameMaker default)

`Structured_GetGeneralNamecase()` returns `True`. The parser converts general names in the DTD to uppercase, meaning that case is not significant for attribute, element, and notation names.

If *NAMECASE GENERAL* is NO, this function returns `False`, meaning that case is significant for attribute, element, and notation names.

When you import a document, FrameMaker looks for the SGML declaration in the following order and locations:

1. An explicit declaration preceding the DTD in the document you import
2. A specification in a subset of your application
3. The default declaration provided as part of FrameMaker

When you export a document, an SGML declaration can be provided as a subset of your application or can be part of the export DTD.

Note that the SGML declaration is specific to the document instance. This means you cannot reliably get the NAMECASE GENERAL parameter until after a begin event for either a book, book component, or document. This is true for both importing and exporting.

Returns

True or False. For XML always returns False.

Examples

The following code retrieves the status of *NAMECASE GENERAL* from an SGML declaration:

```
. . .
BoolT generalIsCapped;
. . .
generalIsCapped = Structured_GetGeneralNamecase();
. . .
```

See also

- [“Structured_GetEntityNamecase\(\)” on page 65](#)
- [“Structured_GetElementDef\(\)” on page 62](#)
- [“Structured_GetNotationDef\(\)” on page 82](#)
- [“Structured_GetFirstElementName\(\)” on page 67](#)
- [“Structured_GetNextElementName\(\)” on page 77](#)
- [“Structured_GetFirstNotationName\(\)” on page 70](#)
- [“Structured_GetNextNotationName\(\)” on page 80](#)
- [“Primitive data types” on page 383](#) for more information on BoolT

Structured_GetLcnmchar()

Deprecated: Sgml_GetLcnmchar()

Gets the *LCNMCHAR* parameter from the declaration for the current SGML document.

Synopsis

```
#include "fm_struct.h"

. . .
ConStringT Structured_GetLcnmchar(VoidT);
```

Arguments

None.

Details

To retrieve the text string describing lowercase name characters from the SGML declaration, call `Structured_GetLcnmchar()`.

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

When you import a document, FrameMaker looks for the SGML declaration in the following order and locations:

1. An explicit declaration preceding the DTD in the document you import
2. A specification in a subset of your application
3. The default declaration provided as part of FrameMaker

When you export a document, an SGML declaration can be provided as a subset of your application or can be part of the export DTD.

Note that the SGML declaration is specific to the document instance. This means you cannot reliably get the *LCNMCHAR* parameter until after a begin event for either a book, book component, or document. This is true for both importing and exporting.

Returns

The *LCNMCHAR* text string, or `NULL` if the *LCNMCHAR* text string is `NULL`. For XML always returns `NULL`.

Examples

The following code retrieves the *LCNMCHAR* text string from the SGML declaration:

```
. . .
ConStringT lcnmcharInfo;
. . .
lcnmcharInfo = Structured_GetLcnmchar();
. . .
```

See also

- [“Structured_GetLcnmstrt\(\)” on page 74](#)
- [“Structured_GetUcnmchar\(\)” on page 89](#)
- [“Structured_GetUcnmstrt\(\)” on page 90](#)
- [“Primitive data types” on page 383](#) for more information on ConStringT

Structured_GetLcnmstrt()

Deprecated: Sgml_GetLcnmstrt()

Gets the *LCNMSTRT* parameter from the declaration for the current document.

Synopsis

```
#include "fm_struct.h"
. . .
ConStringT Structured_GetLcnmchar(VoidT);
```

Arguments

None.

Details

To retrieve the text string describing lowercase name-starting characters from the SGML declaration, call `Structured_GetLcnmstrt()`.

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

When you import a document, FrameMaker looks for the SGML declaration in the following order and locations:

1. An explicit declaration preceding the DTD in the document you import
2. A specification in a subset of your application
3. The default declaration provided as part of FrameMaker

When you export a document, an SGML declaration can be provided as a subset of your application or can be part of the export DTD.

Note that the SGML declaration is specific to the document instance. This means you cannot reliably get the *LCNMSTRT* parameter until after a begin event for either a book, book component, or document. This is true for both importing and exporting.

Returns

The *LCNMSTRT* text string, or `NULL` if the *LCNMSTRT* text string is `NULL`. For XML always returns `NULL`.

Examples

The following code retrieves the *LCNMSTRT* text string from the SGML declaration:

```
. . .
ConStringT lcnmstrtInfo;
. . .
lcnmstrtInfo = Structured_GetLcnmstrt();
. . .
```

See also

- [“Structured_GetLcnmchar\(\)” on page 73](#)
- [“Structured_GetUcnmchar\(\)” on page 89](#)
- [“Structured_GetUcnmstrt\(\)” on page 90](#)
- [“Primitive data types” on page 383](#) for more information on `ConStringT`

Structured_GetLoc()

Deprecated: Sgml_GetLoc()

Reports the location in the markup document that triggered the most recent import event.

Synopsis

```
#include "fm_struct.h"
. . .
VoidT Structured_GetLoc(UIntT *linenop, FilePathT **fpp);
```

Arguments

<i>linenop</i>	Storage location for the document line number where the last processing event was triggered
<i>fpp</i>	Pointer to the file path of the markup file being processed

Details

When importing from markup, it can be useful to know the location in the markup file that triggered the current import event. For example, you might want to log a message that includes a location in the source file.

The parameter *lineop* is a pointer to store the line number from the markup document where the last import event occurred. The parameter **fpp* is a pointer for the markup document's file path, or the file path for a text entity's source file.

Important: Your client must not free **fpp* after calling `Structured_GetLoc()`.

You should only use `Structured_GetLoc()` when importing files. If you call it on export, **fpp* is set to `NULL`, and *lineop* is undefined. Also note that on import you will not get a valid **fpp* for `SR_EVT_BEGIN_READER` events. FrameMaker doesn't build a file path structure for the source file until you actually begin a document or book.

Returns

`VoidT`.

Examples

Following is an import event handler that prints out the line number and filename for the trigger of each event — *except* the begin reader and end reader events.

```
. . .

SrwErrorT Sr_EventHandler(eventp, srObj)
    SrEventT *eventp;
    SrConvObjT srObj;
{
    FilePathT *filep;
    UIntT lineNum;
    StringT s;

    if(eventp->evtype != SR_EVT_BEGIN_READER &&
        eventp->evtype != SR_EVT_END_READER) {
        Structured_GetLoc(&lineNum, &filep);
        s = F_FilePathToPathName(filep, FDefaultPath);
        F_Printf(NULL, "\nEvent for line: %d in file %s",
            lineNum, s);
        /* Free filepath and string. */
        F_ApiDeallocateString(&s);
        F_FilePathFree(filep);
    }
    return (Sr_Convert(eventp, srObj));
}
```

See also

- [“Structured_GetAppinfo\(\)” on page 57](#)
- [“FilePathT” on page 409](#)
- [“Primitive data types” on page 383](#) for more information on UIntT

Structured_GetNextElementName()

Deprecated: Sgml_GetNextElementName()

Retrieves the generic identifier (GI) of the next element in the current DTD for which a declaration is reported by the parser after a specified element.

Synopsis

```
#include "fm_struct.h"
. . .
ConStringT Structured_GetNextElementName(StringT etag);
```

Arguments

etag Name of the preceding element

Details

This function is useful to loop through all elements declared in a DTD; use it in conjunction with `Structured_GetFirstElementName()`. You can pass the name returned by this function to `Structured_GetElementDef()` to retrieve a pointer to the structure containing the element's definition.

etag is the name of the element from which to start searching for the next element.

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

For SGML the reference concrete syntax is case-insensitive for element names, so *etag* can be passed in lowercase, uppercase, or mixed case. For XML, or if the SGML declaration uses naming rules to override the SGML reference concrete syntax, *etag* is case-sensitive.

On import, the DTD is specified as part of the document instance. On export, the DTD is specified within the current structure application.

Returns

The *GI* of the next element in the DTD that follows *etag*.

Examples

The following code retrieves the names of all declared elements in a DTD:

```
. . .
const StructuredElementDefT *elemDefp;
ConStringT elemName;
. . .
switch(eventp->evtype) {
case SR_EVT_BEGIN_DOC:
. . .
    elemName = Structured_GetFirstElementName();
    while (elemName) {
        elemDefp = Structured_GetElementDef((StringT)elemName);
        /* Process the element definition as desired. */
        elemName = Structured_GetNextElementName((StringT)
elemName);
    }
. . .
```

See also

- [“Structured_GetFirstElementName\(\)” on page 67](#)
- [“Structured_GetElementDef\(\)” on page 62](#)
- [“Primitive data types” on page 383](#) for more information on ConStringT and StringT

Structured_GetNextEntityName()

Deprecated: Sgml_GetNextEntityName()

Retrieves the name of the next entity in the current DTD for which a declaration is reported by the parser after a specified entity.

Synopsis

```
#include "fm_struct.h"
. . .
ConStringT Structured_GetNextEntityName(StructuredEntityScopeT
scope,
StringT ename);
```

Arguments

<i>scope</i>	STRUCTURED_ES_GENERAL for a general entity, or STRUCTURED_ES_PARAMETER for a parameter entity
<i>ename</i>	Name of the preceding entity

Details

This function is useful to loop through all the declared entities; use it in conjunction with `Structured_GetFirstEntityName()`. You can pass the name returned by this function to `Structured_GetEntityDef()` to retrieve a pointer to the structure containing the entity's definition.

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

scope indicates the type of parameter to search for. If *scope* is set to `STRUCTURED_ES_PARAMETER`, the entity name returned by this function is preceded by its initial parameter entity reference open (*PERO*) delimiter

ename is the name of the entity from which to start searching for the next entity. Omit the *PERO* delimiter as part of *ename*.

For SGML the reference concrete syntax is case-insensitive for element names, so *ename* can be passed in lowercase, uppercase, or mixed case. For XML, or if the SGML declaration uses naming rules to override the SGML reference concrete syntax, *ename* is case-sensitive.

On import, the DTD is specified as part of the document instance. On export, the DTD is specified within the current structure application.

Returns

The name of the next entity in the DTD that follows *ename*.

Examples

The following code retrieves the definitions of all declared parameter entities in a DTD:

```
. . .
ConStringT entName;
const StructuredEntityDefT *entDefp;
. . .
entName = Structured_GetFirstEntityName(STRUCTURED_ES_PARAMETER);
while (entName)
{
    entDefp = Structured_GetEntityDef(STRUCTURED_ES_PARAMETER,
                                     (StringT)entName);
    /* Now process the entity definition. */
    entName =
    Structured_GetNextEntityName(STRUCTURED_ES_PARAMETER,
                                (StringT)entName);
}
. . .
```

See also

- [“Structured_GetFirstEntityName\(\)” on page 68](#)
- [“Structured_GetEntityDef\(\)” on page 63](#)
- [“StructuredEntityScopeT” on page 388](#)
- [“Primitive data types” on page 383](#) for more information on `ConStringT` and `StringT`

Structured_GetNextNotationName()

Deprecated: `Sgml_GetNextNotationName()`

Retrieves the name of the next data content-notation in the current DTD for which a declaration is reported by the parser after a specified data-content notification.

Synopsis

```
#include "fm_struct.h"
. . .
ConStringT Structured_GetNextNotationName(StringT nname);
```

Arguments

nname Name of the preceding notation

Details

This function is useful to loop through all the declared notations; use it in conjunction with `Structured_GetFirstNotationName()`. You can pass the name returned by this function to `Structured_GetNotationDef()` to retrieve a pointer to the structure containing the notation's definition.

nname is the name of the data-content notation from which to start searching for the next notation.

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

For SGML the reference concrete syntax is case-insensitive for notation names. For XML, or if the SGML declaration uses naming rules to override the SGML reference concrete syntax, notation names are case-sensitive.

On import, the DTD is specified as part of the document instance. On export, the DTD is specified within the current structure application.

Returns

The name of the next data-content notation in the DTD that follows *nname*.

Examples

The following code retrieves the names of all declared data-content notations in a DTD:

```
. . .
ConStringT notateName;
const StructuredNotationDefT *notationp;
. . .
notateName = Structured_GetFirstNotationName();
while (notateName)
{
    notationp = Structured_GetNotationDef((StringT)notateName);
    /* Process the notation definition here */
    notateName = Structured_GetNextNotationName(
        (StringT)notateName);
}
. . .
```

See also

- [“Structured_GetFirstNotationName\(\)” on page 70](#)
- [“Structured_GetNotationDef\(\)” on page 82](#)
- [“Primitive data types” on page 383](#) for more information on ConStringT and StringT

Structured_GetNotationDef()

Deprecated: Sgml_GetNotationDef()

Returns the definition of the specified data-content notation in the DTD.

Synopsis

```
#include "fm_struct.h"

. . .
const StructuredNotationDefT *Structured_GetNotationDef(StringT
nname);
```

Arguments

nname Name of the data-content notation to query

Details

To get a pointer to the structure containing the definition of a specified data-content notation in a DTD, pass the name of the data-content notation to `Structured_GetNotationDef()`.

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

For SGML the reference concrete syntax is case-insensitive for element names, so *nname* can be passed in lowercase, uppercase, or mixed case. For XML, or if the SGML declaration uses naming rules to override the SGML reference concrete syntax, *nname* is case-sensitive.

On import, the DTD is specified as part of the SGML document instance. On export, the DTD is specified within the current structure application.

Returns

A pointer to an `StructuredNotationDefT` structure, or `NULL` if there is no declaration for this notation.

Examples

The following code retrieves the definitions of all declared data-content notations in a DTD:

```
. . .
ConStringT notateName;
const StructuredNotationDefT *notationp;
. . .
notateName = Structured_GetFirstNotationName();
```

```
while (notateName)
{
    notationp = Structured_GetNotationDef((StringT)notateName);
    /* Process the notation definition here */
    notateName = Structured_GetNextNotationName(
        (StringT)notateName);
}
. . .
```

See also

- [“Structured_GetFirstNotationName\(\)” on page 70](#)
- [“Structured_GetNextNotationName\(\)” on page 80](#)
- [“StructuredNotationDefT” on page 415](#)
- [“Primitive data types” on page 383](#) for more information on `StringT`

Structured_GetOptionalFeature()

Deprecated: Sgml_GetOptionalFeature()

Tests whether a specified optional feature is both supported by FrameMaker and used in the SGML declaration.

Synopsis

```
#include "fm_struct.h"
. . .
IntT Structured_GetOptionalFeature(StructuredFeatureTypeT
feature);
```

Arguments

feature The type of feature for which to query

Details

The *feature* argument is of the type `StructuredFeatureTypeT`, which is defined in the following table. The table also shows which features FrameMaker can process.

	Value	Meaning
Processing supported by FrameMaker	STRUCTURED_FEAT_SHORTREF	<i>SHORTREF</i>
	STRUCTURED_FEAT_OMITTAG	<i>OMITTAG</i>
	STRUCTURED_FEAT_SHORTTAG	<i>SHORTTAG</i>
	STRUCTURED_FEAT_FORMAL	<i>FORMAL</i>

	Value	Meaning
Processing not supported by FrameMaker	STRUCTURED_FEAT_DATATAG	<i>DATATAG</i>
	STRUCTURED_FEAT_RANK	<i>RANK</i>
	STRUCTURED_FEAT_SIMPLE_LINK	<i>SIMPLE</i>
	STRUCTURED_FEAT_IMPLICIT_LINK	<i>IMPLICIT</i>
	STRUCTURED_FEAT_EXPLICIT_LINK	<i>EXPLICIT</i>
	STRUCTURED_FEAT_CONCUR	<i>CONCUR</i>
	STRUCTURED_FEAT_SUBDOC	<i>SUBDOC</i>

If this function returns 1, an optional feature is used in the SGML declaration and can be processed by your application.

If the function returns 0, the feature is either not used in the SGML declaration, or is unsupported by FrameMaker. If an unsupported optional feature is encountered in a declaration or document during processing, an error is issued, and processing is terminated.

When you import a document, FrameMaker looks for the SGML declaration in the following order and locations:

- 1.An explicit declaration preceding the DTD in the document you import
- 2.A specification in a subset of your application
- 3.The default declaration provided as part of FrameMaker

When you export a document, an SGML declaration can be provided as a subset of your application or can be part of the export DTD.

Returns

0 for a feature that is not supported in the SGML declaration or that is unsupported by FrameMaker, or 1 for a supported feature. For XML always returns 0.

Examples

The following code checks to see if the SGML optional feature *SHORTREF* is supported in an SGML declaration:

```
. . .
IntT shortTagSupported;
. . .
shortTagSupported =
Structured_GetOptionalFeature( STRUCTURED_FEAT_SHORTTAG );
. . .
```

See also

- [“Structured_GetAppinfo\(\)” on page 57](#)
- [“Structured_GetDelimiterString\(\)” on page 59](#)

- [“Structured_GetDocTypeName\(\)” on page 61](#)
- [“Structured_GetQuantity\(\)” on page 85](#)
- [“Structured_GetReservedName\(\)” on page 87](#)
- [“StructuredFeatureTypeT” on page 390](#)
- [“Primitive data types” on page 383](#) for more information on `BoolT`

Structured_GetQuantity()

Deprecated: `Sgml_GetQuantity()`

Returns the numeric limit for the specified quantity type.

Synopsis

```
#include "fm_struct.h"
. . .
UIntT Structured_GetQuantity(StructuredQuantityTypeT quantity);
```

Arguments

quantity Quantity type for which to query

Details

To discover the quantity limit for a specific quantity name as specified in an SGML declaration, use `Structured_GetQuantity()`. In XML and SGML, quantity limits are numeric restrictions on some aspect of markup, such as the maximum length of a name or name token. This function is useful for checking quantity assignments in the declared concrete syntax to see if they differ from the default quantities assigned in the quantity set of the standard SGML reference concrete syntax.

quantity is of the type `StructuredQuantityTypeT`, which is defined as follows:

Value	Meaning
<code>STRUCTURED_QTY_ATT CNT</code>	Maximum number of attribute names and name tokens in an attribute definition list
<code>STRUCTURED_QTY_ATT SPLEN</code>	Maximum normalized length of an attribute specification list for a start-tag
<code>STRUCTURED_QTY_BSE QLEN</code>	Maximum length of a blank sequence in a short reference string
<code>STRUCTURED_QTY_DTAGLEN</code>	Maximum length of a data tag
<code>STRUCTURED_QTY_DTE MPLEN</code>	Maximum length of a data-tag template or pattern template
<code>STRUCTURED_QTY_ENTLVL</code>	Maximum nesting level for entities
<code>STRUCTURED_QTY_GRP CNT</code>	Maximum number of tokens in a group

Value	Meaning
STRUCTURED_QTY_GRPGETCNT	Maximum number of content tokens at all levels of a content model
STRUCTURED_QTY_GRPPLVL	Maximum nesting level of model groups
STRUCTURED_QTY_LITLEN	Maximum length of a parameter literal or an attribute literal
STRUCTURED_QTY_NAMELEN	Maximum length of a name, a name token, a number, or other item
STRUCTURED_QTY_NORMSEP	Length to use instead of counting separators when calculating normalized lengths
STRUCTURED_QTY_PILEN	Maximum length of a processing instruction
STRUCTURED_QTY_TAGLEN	Maximum length of a start-tag
STRUCTURED_QTY_TAGLVL	Maximum nesting level of open elements

When you import a document, FrameMaker looks for the SGML declaration in the following order and locations:

1. An explicit declaration preceding the DTD in the document you import
2. A specification in a subset of your application
3. The default declaration provided as part of FrameMaker

When you export a document, an SGML declaration can be provided as a subset of your application or can be part of the export DTD.

Returns

An integer containing the quantity limit for the specified type. For XML always returns 0.

Examples

The following code returns the quantity limit for the maximum number of attribute names and name tokens that can appear in an attribute definition list:

```
. . .
UIntT nameLen;
. . .
nameLen = Structured_GetQuantity(STRUCTURED_QTY_NAMELEN);
. . .
```

See also

- [“Structured_GetAppinfo\(\)” on page 57](#)
- [“Structured_GetDelimiterString\(\)” on page 59](#)
- [“Structured_GetDocTypeName\(\)” on page 61](#)
- [“Structured_GetOptionalFeature\(\)” on page 83](#)

- [“Structured_GetReservedName\(\)” on page 87](#)
- [“StructuredQuantityTypeT” on page 391](#)
- [“Primitive data types” on page 383](#) for more information on UIntT

Structured_GetReservedName()

Deprecated: Sgml_GetReservedName()

Returns the reserved name from the SGML declaration for a specified reserved name type.

Synopsis

```
#include "fm_struct.h"
. . .
ConStringT Structured_GetReservedName(StructuredReservedNameT
rname);
```

Arguments

rname Reserved name to return

Details

To examine SGML reserved names in an SGML declaration to see if they differ from the defaults in the reference concrete syntax, call `Structured_GetReservedName()`. FrameMaker does not support reserved names that differ from the SGML reference concrete syntax.

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

rname is an enumerated type corresponding to a standard reserved word in reference concrete syntax, or to a reserved name that can occur only in the SGML declaration. The following values are enumerated for `StructuredReservedNameT`—entries with a "†" indicate values that are valid for both SGML and XML:

STRUCTURED_RN_ANY †	STRUCTURED_RN_INCLUDE †	STRUCTURED_RN_POSTLINK
STRUCTURED_RN_ATTLIST †	STRUCTURED_RN_INITIAL	STRUCTURED_RN_PUBLIC †
STRUCTURED_RN_CDATA †	STRUCTURED_RN_LINK	STRUCTURED_RN_RCDATA
STRUCTURED_RN_CONREF	STRUCTURED_RN_LINKTYPE	STRUCTURED_RN_RE
STRUCTURED_RN_CURRENT	STRUCTURED_RN_MD	STRUCTURED_RN_REQUIRED †
STRUCTURED_RN_DEFAULT	STRUCTURED_RN_MS	STRUCTURED_RN_RESTORE
STRUCTURED_RN_DOCTYPE †	STRUCTURED_RN_NAME	STRUCTURED_RN_RS
STRUCTURED_RN_ELEMENT †	STRUCTURED_RN_NAMES	STRUCTURED_RN_SDATA
STRUCTURED_RN_EMPTY †	STRUCTURED_RN_NDATA †	STRUCTURED_RN_SHORTREF

STRUCTURED_RN_ENDTAG	STRUCTURED_RN_NMTOKEN †	STRUCTURED_RN_SIMPLE
STRUCTURED_RN_ENTITIES †	STRUCTURED_RN_NMTOKENS †	STRUCTURED_RN_SPACE
STRUCTURED_RN_ENTITY †	STRUCTURED_RN_NOTATION †	STRUCTURED_RN_STARTTAG
STRUCTURED_RN_FIXED †	STRUCTURED_RN_NUMBER	STRUCTURED_RN_SUBDOC
STRUCTURED_RN_ID †	STRUCTURED_RN_NUMBERS	STRUCTURED_RN_SYSTEM †
STRUCTURED_RN_IDLINK	STRUCTURED_RN_NUTOKEN	STRUCTURED_RN_TEMP
STRUCTURED_RN_IDREF †	STRUCTURED_RN_NUTOKENS	STRUCTURED_RN_USELINK
STRUCTURED_RN_IDREFS †	STRUCTURED_RN_O	STRUCTURED_RN_USEMAP
STRUCTURED_RN_IGNORE †	STRUCTURED_RN_PCDATA †	
STRUCTURED_RN_IMPLIED †	STRUCTURED_RN_PI	

When you import a document, FrameMaker looks for the SGML declaration in the following order and locations:

- 1.An explicit declaration preceding the DTD in the document you import
- 2.A specification in a subset of your application
- 3.The default declaration provided as part of FrameMaker

When you export a document, an SGML declaration can be provided as a subset of your application or can be part of the export DTD.

Returns

A string containing the requested reserved name.

Examples

The following example returns the reserved name specified in the SGML declaration for the reference concrete syntax reserved word *DOCTYPE*:

```
. . .
ConStringT resName;
. . .
resName = Structured_GetReservedName( STRUCTURED_RN_DOCTYPE );
. . .
```

See also

- [“Structured_GetAppinfo\(\)” on page 57](#)
- [“Structured_GetDelimiterString\(\)” on page 59](#)
- [“Structured_GetDocTypeName\(\)” on page 61](#)
- [“Structured_GetOptionalFeature\(\)” on page 83](#)
- [“Structured_GetQuantity\(\)” on page 85](#)
- [“StructuredReservedNameT” on page 392](#)

- [“Primitive data types” on page 383](#) for more information on `ConStringT`

Structured_GetUcnmchar()

Deprecated: `Sgml_GetUcnmchar()`*Sgml_GetUcnmchar()*

Gets the *UCNMCHAR* parameter from the declaration for the current markup document.

Synopsis

```
#include "fm_struct.h"

. . .

ConStringT Structured_GetUcnmchar(VoidT);
```

Arguments

None.

Details

To retrieve the text string describing uppercase name characters from the SGML declaration, call `Structured_GetUcnmchar()`.

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

When you import a document, FrameMaker looks for the SGML declaration in the following order and locations:

1. An explicit declaration preceding the DTD in the document you import
2. A specification in a subset of your application
3. The default declaration provided as part of FrameMaker

When you export a document, an SGML declaration can be provided as a subset of your application or can be part of the export DTD.

Note that the SGML declaration is specific to the document instance. This means you cannot reliably get the *UCNMCHAR* parameter until after a begin event for either a book, book component, or document. This is true for both importing and exporting.

Returns

The *UCNMCHAR* text string, or `NULL` if the *UCNMCHAR* text string is `NULL`. For XML always returns `NULL`.

Examples

The following code retrieves the *UCNMCHAR* text string from the SGML declaration:

```
. . .
ConStringT ucnmcharInfo;
. . .
ucnmcharInfo = Structured_GetUcnmchar();
. . .
```

See also

- [“Structured_GetUcnmstrt\(\)” on page 90](#)
- [“Structured_GetLcnmchar\(\)” on page 73](#)
- [“Structured_GetLcnmstrt\(\)” on page 74](#)
- [“Primitive data types” on page 383](#) for more information on ConStringT

Structured_GetUcnmstrt()

Deprecated: *Sgml_GetUcnmstrt()**Sgml_GetUcnmstrt()*

Gets the *UCNMSTRT* parameter from the declaration for the current markup document.

Synopsis

```
#include "fm_struct.h"
. . .
ConStringT Structured_GetUcnmstrt(VoidT);
```

Arguments

None.

Details

To retrieve the text string describing uppercase name-starting characters from the SGML declaration, call *Structured_GetUcnmstrt()*.

Important: The pointer returned by this function references an internal data structure. No attempt should be made to modify the contents of that structure, or to free it.

When you import a document, FrameMaker looks for the SGML declaration in the following order and locations:

1. An explicit declaration preceding the DTD in the document you import
2. A specification in a subset of your application
3. The default declaration provided as part of FrameMaker

When you export a document, an SGML declaration can be provided as a subset of your application or can be part of the export DTD.

Note that the SGML declaration is specific to the document instance. This means you cannot reliably get the *UCNMSTRT* parameter until after a begin event for either a book, book component, or document. This is true for both importing and exporting.

Returns

The *UCNMSTRT* text string, or `NULL` if the *UCNMSTRT* text string is `NULL`. For XML always returns `NULL`.

Examples

The following code retrieves the *UCNMSTRT* text string from the SGML declaration:

```
. . .
ConStringT ucnmstrtInfo;
. . .
ucnmstrtInfo = Structured_GetUcnmstrt();
. . .
```

See also

- [“Structured_GetUcnmchar\(\)” on page 89](#)
- [“Structured_GetLcnmchar\(\)” on page 73](#)
- [“Structured_GetLcnmstrt\(\)” on page 74](#)
- [“Primitive data types” on page 383](#) for more information on `ConStringT`

Structured_IsAttrCDATA()

Deprecated: `Sgml_IsAttrCDATA()`*Sgml_IsAttrCDATA()*

Tests whether an attribute is a *CDATA* attribute.

Synopsis

```
#include "fm_struct.h"
. . .
BoolT Structured_IsAttrCDATA(StructuredAttrValT *attrVal);
```

Arguments

attrVal Pointer to the attribute to query

Details

To determine if an attribute is a *CDATA* attribute, use the `Structured_IsAttrCDATA()` macro. It tests the `sgmlAttrFlags` field of *attrVal* to see if the `STRUCTURED_ATTR_IS_CDATA` flag is set.

Attributes are specified in the DTD. On import, the DTD is specified as part of the document instance. On export, the DTD is specified within the current structure application.

Returns

1 if the attribute is a *CDATA* attribute; otherwise, 0.

Examples

The following code would be found in an export event handler. It tests the `sgmlAttrFlags` field of an attribute for its type, and prints out the appropriate description. Note that not all of attribute type flags can be set for a single attribute. Also note how the code loops through all the attributes of the given element.

```
. . .
StructuredAttrValsT attrVals;
StringT elemGi;
. . .
switch(eventp->evtype) {
    case SW_EVT_BEGIN_ELEM:
        elemGi = Sw_GetStructuredGi(swObj);
        attrVals = Sw_GetAttrVals(swObj);
        for(i=0; i<attrVals.len; i++) {
            if(Structured_IsAttrCDATA(&attrVals.val[i])){
                F_Printf(NULL, "\nElement %s, attribute %s is CDATA",
                    elemGi, attrVals.val[i].sgmlAttrName);
            }
            if(Structured_IsAttrFixed(&attrVals.val[i])){
                F_Printf(NULL, "\nElement %s, attr %s is fixed",
                    elemGi, attrVals.val[i].sgmlAttrName);
            }
            if(Structured_IsAttrNameToken(&attrVals.val[i])){
                F_Printf(NULL, "\nElement %s, attr %s is NMTOKEN",
                    elemGi, attrVals.val[i].sgmlAttrName);
            }
            if(Structured_IsAttrValSpecified(&attrVals.val[i])){
                F_Printf(NULL, "\nElement %s, attr %s is specified",
                    elemGi, attrVals.val[i].sgmlAttrName);
            }
            if(Structured_IsIdAttr(&attrVals.val[i])){
                F_Printf(NULL, "\nElement %s, attr %s is ID",
                    elemGi, attrVals.val[i].sgmlAttrName);
            }
        }
    }
```

```
        if(Structured_IsIdRefAttr(&attrVals.val[i])){
            F_Printf(NULL, "\nElement %s, attr %s is IDREF",
                elemGi, attrVals.val[i].vt);
        }
    }
    F_ApiDeallocateString(&elemGi);
    Structured_DeallocateAttrVals(&attrVals);
    . . .
```

See also

- [“Structured_IsAttrFixed\(\)” on page 93](#)
- [“Structured_IsAttrNameToken\(\)” on page 94](#)
- [“Structured_IsAttrValSpecified\(\)” on page 95](#)
- [“Structured_IsIdAttr\(\)” on page 96](#)
- [“Structured_IsIdRefAttr\(\)” on page 97](#)
- [“StructuredAttrValT” on page 412](#)
- [“Primitive data types” on page 383](#) for more information on `BoolT`

Structured_IsAttrFixed()

Deprecated: `Sgml_IsAttrFixed()`*Sgml_IsAttrFixed()*

Tests whether an attribute is a fixed attribute.

Synopsis

```
#include "fm_struct.h"
. . .
BoolT Structured_IsAttrFixed(StructuredAttrValT *attrVal);
```

Arguments

attrVal Pointer to the attribute to query

Details

To determine if an attribute is a fixed attribute, one whose specified value cannot differ from its default value, use the `Structured_IsAttrFixed()` macro. It tests the `sgmlAttrFlags` field of *attrVal* to see if the `STRUCTURED_ATTR_IS_FIXED` flag is set.

Attributes are specified in the DTD. On import, the DTD is specified as part of the document instance. On export, the DTD is specified within the current structure application.

Returns

1 if the attribute is a fixed attribute; otherwise, 0.

Examples

See the example for [“Structured_IsAttrCDATA\(\)” on page 91](#).

See also

- [“Structured_IsAttrCDATA\(\)” on page 91](#)
- [“Structured_IsAttrNameToken\(\)” on page 94](#)
- [“Structured_IsAttrValSpecified\(\)” on page 95](#)
- [“Structured_IsIdAttr\(\)” on page 96](#)
- [“Structured_IsIdRefAttr\(\)” on page 97](#)
- [“StructuredAttrValT” on page 412](#)
- [“Primitive data types” on page 383](#) for more information on `BoolT`

Structured_IsAttrNameToken()

Deprecated: `Sgml_IsAttrNameToken()` *Sgml_IsAttrNameToken()*

Tests whether an attribute has a name token group for its declared value.

Synopsis

```
#include "fm_struct.h"
. . .
BoolT Structured_IsAttrNameToken(StructuredAttrValT *attrVal);
```

Arguments

attrVal Pointer to the attribute to query

Details

To determine if an attribute has a name token group for its declared value, use the `Structured_IsAttrNameToken()` macro. It tests the `sgmlAttrFlags` field of `attrVal` to see if the `STRUCTURED_ATTR_IS_NAME_TOKEN` flag is set.

Attributes are specified in the DTD. On import, the DTD is specified as part of the document instance. On export, the DTD is specified within the current structure application.

Returns

1 if the attribute has a name token group for its declared value, or 0 if it is a notation attribute.

Examples

See the example for [“Structured_IsAttrCDATA\(\)” on page 91](#).

See also

- [“Structured_IsAttrCDATA\(\)” on page 91](#)

- [“Structured_IsAttrFixed\(\)” on page 93](#)
- [“Structured_IsAttrValSpecified\(\)” on page 95](#)
- [“Structured_IsIdAttr\(\)” on page 96](#)
- [“Structured_IsIdRefAttr\(\)” on page 97](#)
- [“StructuredAttrValT” on page 412](#)
- [“Primitive data types” on page 383](#) for more information on `BoolT`

Structured_IsAttrValSpecified()

Deprecated: `Sgml_IsAttrValSpecified()`*Sgml_IsAttrValSpecified()*

Tests whether a value is specified for an attribute.

Synopsis

```
#include "fm_struct.h"
. . .
BoolT Structured_IsAttrValSpecified(StructuredAttrValT *attrVal);
```

Arguments

attrVal Pointer to the attribute to query

Details

To determine if a value is specified for an attribute, call the `Structured_IsAttrValSpecified()` macro. It tests the `sgmlAttrFlags` field of *attrVal* to see if the `STRUCTURED_ATTR_VAL_SPECIFIED` flag is set.

Attributes are specified in the DTD. On import, the DTD is specified as part of the document instance. On export, the DTD is specified within the current structure application.

Returns

1 if a value is specified for the attribute, or 0 if it can default.

Examples

See the example for [“Structured_IsAttrCDATA\(\)” on page 91](#).

See also

- [“Structured_IsAttrCDATA\(\)” on page 91](#)
- [“Structured_IsAttrFixed\(\)” on page 93](#)
- [“Structured_IsAttrNameToken\(\)” on page 94](#)
- [“Structured_IsIdAttr\(\)” on page 96](#)
- [“Structured_IsIdRefAttr\(\)” on page 97](#)

- [“StructuredAttrValT” on page 412](#)
- [“Primitive data types” on page 383](#) for more information on `BoolT`

Structured_IsIdAttr()

Deprecated: `Sgml_IsIdAttr()`*Sgml_IsIdAttr()*

Tests if the declared value of an attribute is *ID*.

Synopsis

```
#include "fm_struct.h"
. . .
BoolT Structured_IsIdAttr(StructuredAttrValT *attrVal);
```

Arguments

attrVal Pointer to the attribute to query

Details

To determine if the declared value for an attribute is *ID*, use the `Structured_IsIdAttr()` macro. It tests the `sgmlAttrFlags` field of *attrVal* to see if the `STRUCTURED_ATTR_IS_ID` flag is set.

Attributes are specified in the DTD. On import, the DTD is specified as part of the document instance. On export, the DTD is specified within the current structure application.

Returns

1 if the attribute's declared value is *ID*; otherwise, 0.

Examples

See the example for [“Structured_IsAttrCDATA\(\)” on page 91](#).

See also

- [“Structured_IsAttrCDATA\(\)” on page 91](#)
- [“Structured_IsAttrFixed\(\)” on page 93](#)
- [“Structured_IsAttrNameToken\(\)” on page 94](#)
- [“Structured_IsAttrValSpecified\(\)” on page 95](#)
- [“Structured_IsIdRefAttr\(\)” on page 97](#)
- [“StructuredAttrValT” on page 412](#)
- [“Primitive data types” on page 383](#) for more information on `BoolT`

Structured_IsIdRefAttr()

Deprecated: `Sgml_IsIdRefAttr()`*Sgml_IsIdRefAttr()*

Tests whether an attribute is an *IDREF* attribute.

Synopsis

```
#include "fm_struct.h"

. . .

BoolT Structured_IsIdRefAttr(StructuredAttrValT *attrVal);
```

Arguments

attrVal Pointer to the attribute to query

Details

To determine if an attribute is a *IDREF* attribute, use the `Structured_IsIdRefAttr()` macro. It tests the `sgmlAttrFlags` field of *attrVal* to see if the `STRUCTURED_ATTR_IS_IDREF` flag is set.

Attributes are specified in the DTD. On import, the DTD is specified as part of the document instance. On export, the DTD is specified within the current structure application.

Returns

1 if the attribute is a *IDREF* attribute; otherwise, 0.

Examples

See the example for [“Structured_IsAttrCDATA\(\)” on page 91](#).

See also

- [“Structured_IsAttrCDATA\(\)” on page 91](#)
- [“Structured_IsAttrFixed\(\)” on page 93](#)
- [“Structured_IsAttrNameToken\(\)” on page 94](#)
- [“Structured_IsAttrValSpecified\(\)” on page 95](#)
- [“Structured_IsIdAttr\(\)” on page 96](#)
- [“StructuredAttrValT” on page 412](#)
- [“Primitive data types” on page 383](#) for more information on `BoolT`

Sr_AddTableRows()

Adds one or more rows to an existing FrameMaker table during the import process.

Synopsis

```
#include "fm_struct.h"

. . .

SrwErrorT Sr_AddTableRows(F_ObjHandleT docId,
    F_ObjHandleT refRowId, IntT direction, IntT count);
```

Arguments

<i>docId</i>	ID of the document containing the table
<i>refRowId</i>	ID of the row at which to start adding rows
<i>direction</i>	Direction from <i>refRowId</i> in which to add rows
<i>count</i>	Number of rows to add

Details

To add one or more rows to a FrameMaker table previously inserted into a FrameMaker document during the import process (most likely as the result of converting an SR_OBJ_TABLE conversion object), call *Sr_AddTableRows()*. If you add rows in the middle of a straddle, *Sr_AddTableRows()* automatically extends that straddle. If you add rows at the end of straddle, *Sr_AddTableRows()* does not automatically extend the straddle.

direction is a constant that can have one of the following values:

Direction constant	Meaning
FV_Above	Add rows above row specified by <i>refRowId</i>
FV_Below	Add rows below row specified by <i>refRowId</i>
FV_Body	Add body rows at bottom of existing body rows
FV_Footing	Add footing rows at bottom of existing footing rows
FV_Heading	Add heading rows at bottom of existing heading rows

If *direction* is FV_Above or FV_Below, rows added are the same type as the row specified by *refRowId*. If *direction* is FV_Body, FV_Footing, or FV_Heading, *refRowId* is ignored.

Returns

On error, SRW_E_FAILURE.

Examples

The following code adds two rows after the first row in the selected table:

```
. . .
F_ObjHandleT docId, rowId, tableId;
SrwErrorT errorStatus;

. . .
/* Get the document and table IDs. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tableId = F_ApiGetId(FV_SessionId, docId, FP_SelectedTbl);
/* Get the ID for the first row in the table. */
rowId = F_ApiGetId(docId, tblId, FP_FirstRowInTbl);
. . .
/* Add two rows. */
errorStatus = Sr_AddTableRows(docId, rowId, FV_Below, 2);
. . .
```

See also

- [“Sr_RowInUse\(\)” on page 182](#)
- [“Sr_SetTableRowUsed\(\)” on page 227](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on F_ObjHandleT and IntT

Sr_CancelCurBatchFile()

When importing markup files in batch mode, terminates import of the current document and continues processing the next document.

Synopsis

```
#include "fm_struct.h"

. . .
VoidT Sr_CancelCurBatchFile(VoidT);
```

Arguments

None.

Details

This function stops importing the current document at the end of the current event, and allows batch processing to resume with the next file. It only works when the current session is in batch mode. The current batch mode is indicated in the `SrSessionPropsT` structure; you can get the value of this property for the current session using `Sr_GetSessionProps()`.

A batch import process is one that operates on an entire directory of markup files. For example, users can start a batch process by choosing Convert Structured Documents from the File > Utilities menu. Unix users can also start a batch process from the command line.

Use this function when you encounter an error condition, for example, if you cannot find a specified entity.

To cancel import of all markup documents during batch processing, call `Sr_CancelOperation()`.

Returns

`VoidT`.

Examples

The following code cancels processing of the current document during a batch operation if you cannot find the path to a proposed external entity:

```
. . .
FilePathT *entFilePath;
StringT entityName;
. . .
entFilePath = Sr_GetExtEntityFilePath(entityName);
if (!entFilePath)
{
    Sr_CancelCurBatchFile();
    return (SRW_E_FAILURE);
}
. . .
```

See also

- [“Sr_CancelOperation\(\)” on page 102](#)
- [“Sr_GetSessionProps\(\)” on page 167](#) for information on getting the current batch mode
- [“Primitive data types” on page 383](#) for more information on `VoidT`

Sr_CancelOperation()

When importing in batch mode, terminates the import of all markup documents and returns control to FrameMaker. When importing a single file, terminates the import and returns control to FrameMaker.

Synopsis

```
#include "fm_struct.h"
. . .
VoidT Sr_CancelOperation(VoidT);
```

Arguments

None.

Details

This function stops importing the current markup document at the end of the current event, and returns control to FrameMaker. You can call this function when importing a single file, or when importing in batch mode. If the current session is in batch mode, a number of files may have already been imported; this function does not affect any documents that were already imported by the batch process. The current batch mode is indicated in the `SrSessionPropsT` structure; you can get the value of this property for the current session using `Sr_GetSessionProps()`.

Use this function when you encounter an error condition, for example, if you cannot find a specified entity.

Returns

VoidT.

Examples

The following code cancels importing of markup documents if you cannot find the path to a proposed external entity:

```
. . .
FilePathT *entFilePath;
StringT entityName;
. . .
entFilePath = Sr_GetExtEntityFilePath(entityName);
if (!entFilePath)
{
    Sr_CancelOperation();
    return (SRW_E_FAILURE);
}
F_FilePathFree(entFilePath);
. . .
```

See also

- [“Sr_CancelCurBatchFile\(\)” on page 100](#)
- [“Sr_GetSessionProps\(\)” on page 167](#) for information on getting the current batch mode
- [“Primitive data types” on page 383](#) for more information on VoidT

Sr_CellInUse()

Tests whether a specified cell in a FrameMaker table is marked as used.

Synopsis

```
#include "fm_struct.h"

. . .

BoolT Sr_CellInUse(F_ObjHandleT docId, F_ObjHandleT cellId);
```

Arguments

docId ID of the document containing the table

cellId ID of the cell in the table

Details

To test whether a cell in a FrameMaker table is marked as used, call `Sr_CellInUse()`. Internally, FrameMaker uses this function to determine which cell the current insert location should be in.

To mark a cell as used, call `Sr_SetTableCellUsed()`.

Returns

1 if the cell is marked as used; otherwise, 0. If *cellId* is not a cell object, `SRW_errno` is set to `SRW_E_FAILURE` and the function returns 0.

Examples

The following code tests the first cell of the first row in a selected FrameMaker table to see if it is marked as used:

```
. . .
F_ObjHandleT docId, cellId, rowId, tableId;
BoolT cellIsUsed;
. . .
/* Get the document and table IDs. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tableId = F_ApiGetId(FV_SessionId, docId, FP_SelectedTbl);

/* Get the ID for the first row in the table. */
rowId = F_ApiGetId(docId, tableId, FP_FirstRowInTbl);
```

Sr_Convert()

```
/* Get the ID for the first cell of the first row. */
cellId = F_ApiGetId(docId, rowId, FP_FirstCellInRow);
. . .
/* Test the cell to see if it's marked in use. */
cellIsUsed = Sr_CellInUse(docId, cellId);
. . .
```

See also

- [“Sr_SetTableCellUsed\(\)” on page 226](#)
- [“Sr_RowInUse\(\)” on page 182](#)
- [“Sr_SetTableRowUsed\(\)” on page 227](#)
- [“Primitive data types” on page 383](#) for more information on `F_ObjHandleT` and `BoolT`

Sr_Convert()

Converts a specified conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_Convert(SrEventT *eventp, SrConvObjT srObj);
```

Arguments

<i>eventp</i>	Pointer to conversion event to perform
<i>srObj</i>	Conversion object to convert

Details

Call `Sr_Convert()` from a custom import client to convert a conversion object to the FrameMaker document. A client should call this function once for each event/conversion object pair unless the client writes its own changes directly to the FrameMaker document for a given event/conversion object pair.

eventp is the event passed from FrameMaker to your client. It specifies the conversion event to carry out.

srObj is the modified or unmodified conversion object originally passed from FrameMaker to your client. It is the object to convert.

Returns

On error, `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code sends the current event/conversion object pair, generated by FrameMaker during the import process, to the default conversion routine:

```
. . .  
SrErrorT errorStatus;  
. . .  
errorStatus = Sr_Convert(eventp, srObj);  
. . .
```

See also

- [“Sr_EventHandler\(\)” on page 107](#)
- [“SrConvObjT” on page 415](#)
- [“SrEventT” on page 416](#)
- [“SrErrorT” on page 396](#)

Sr_DeallocateSessionProps()

Deletes an import session properties structure and frees the memory allocated for it.

Synopsis

```
#include "fm_struct.h"  
. . .  
VoidT Sr_DeallocateSessionProps(SrSessionPropsT *sessionProps);
```

Arguments

sessionProps Pointer to session properties structure to free

Details

To delete an import session properties structure and free the memory allocated for the structure, call `Sr_DeallocateSessionProps()`.

Import session properties include settings indicating batch mode status, file overwrite permissions, and ruling style for tables. If you call `Sr_GetSessionProps()` to examine these settings, that function returns an import session properties structure. Deallocate this structure with `Sr_DeallocateSessionProps()` when you are done with it. If you call `Sr_SetSessionProps()` to specify import session properties, you must first create and populate an import session properties structure to pass as an argument to the function. When you no longer need that structure, you should deallocate it with `Sr_DeallocateSessionProps()`.

Returns

VoidT.

Examples

The following case statement prints out the import session properties. Finally, the code frees the space allocated to the `SrSessionPropsT` structure:

```
. . .
SrSessionPropsT importSessionProps;
SrConvObjT sessionObj;
. . .
case SR_EVT_BEGIN_READER:
    sessionObj = Sr_GetCurConvObjOfType(SR_OBJ_SESSION);
    importSessionProps = Sr_GetSessionProps(sessionObj);
    if(importSessionProps.isBatchMode)
        F_Printf(NULL, "\nImporting in batch mode");
    else
        F_Printf(NULL, "\nNOT importing in batch mode");
    if(importSessionProps.overwriteFiles)
        F_Printf(NULL, "\nOverwriting files");
    else
        F_Printf(NULL, "\nNOT overwriting files");
    F_Printf(NULL, "\nTable ruling style: %s",
            importSessionProps.tableRulingStyle);
    Sr_DeallocateSessionProps(&importSessionProps);
    break;
. . .
```

See also

- [“Sr_GetSessionProps\(\)” on page 167](#)
- [“Sr_SetSessionProps\(\)” on page 222](#)
- [“SrSessionPropsT” on page 417](#)
- [“Primitive data types” on page 383](#) for more information on `VoidT`

Sr_EventHandler()

Responds to event/conversion pairs passed from FrameMaker to a custom import client. *Sr_EventHandler()* is the user-defined entry point for all structure import/export API client import conversion applications.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_EventHandler(SrEventT *eventp, SrConvObjT srObj);
```

Arguments

<i>eventp</i>	Pointer to conversion event to perform
<i>srObj</i>	Conversion object to convert

Details

Sr_EventHandler() is a user-defined callback function that is the entry-point for all custom import clients. It enables you to modify the behavior of import behavior beyond the modifications permitted through FrameMaker read/write rules. All structure import/export API import clients must define *Sr_EventHandler()*.

Your *Sr_EventHandler()* function should include code for all import modifications you want to make.

Returns

SRW_E_FAILURE on error; SRW_E_SUCCESS on success.

Examples

The following code illustrates how an event handler might be set up to modify column-width specifications in a *CALS* table:

```
. . .
SrwErrorT Sr_EventHandler(SrEventT *eventp, SrConvObjT srObj)
{
    SrwColSpecsT colspecs;
    SrwColSpecT colspec;
    SrConvObjT tblObj;
    switch(Sr_GetObjType(srObj))
    {
        case SR_OBJ_TABLE_ROW:
            /* Retrieve the current table object. */
            tblObj = Sr_GetCurConvObjOfType(SR_OBJ_TABLE);
            /* Retrieve copy of colspecs for table object. */
            colspecs = Sr_GetColSpecs(tblObj);
```

```
        /* If there are no colspecs, exit gracefully. */
        if(colspecs.len == 0)
            break;
        /* Retrieve copy of colspec for the first column. */
        colspec = Srw_GetColSpecByColNum(&colspecs, 0);
        if(SRW_errno == SRW_E_SUCCESS)
        {
            /* Set the column width to 12 cm. */
            F_Free(colspec.width);
            colspec.width = F_StrCopyString("12cm");
            colspec.valueSet |= SRW_COLSPEC_COLWIDTH;
            /* Put new colspec back into list. */
            Srw_SetColSpec(&colspecs, &colspec);
            /* Put the new list back into table object. */
            Sr_SetColSpecs(tblObj, &colspecs);
        }
        /* Free any allocated memory. */
        Srw_DeallocateColSpec(&colspec);
        Srw_DeallocateColSpecs(&colspecs);
        break;
    /* Handle additional objects below (code omitted). */
    . . .
}
/* Convert object and return control to FrameMaker. */
return Sr_Convert(eventp, srObj);
}
```

See also

- [“Sr_Convert\(\)” on page 104](#)
- [“SrConvObjT” on page 415](#)
- [“SrEventT” on page 416](#)
- [“SrwErrorT” on page 396](#)

Sr_GetAssociatedEvent()

Returns an import event associated with a specified import conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
SrEventT *Sr_GetAssociatedEvent(SrConvObjT srObj);
```

Arguments

srObj Conversion object for which to return an event

Details

When an import conversion object is created, it and its related event are pushed onto the conversion object stack. *Sr_GetAssociatedEvent()* retrieves a pointer to the event associated with a specific conversion object. The event structure describes the data in the markup document that triggered the event.

This function is often used in conjunction with *Sr_GetCurConvObjOfType()* to retrieve information about a previously processed event/conversion object pair. For example, you might want to know the element tag for the parent of the current element. To do that, you get the parent conversion object, and then get the tag from its associated event.

Returns

A pointer to the event data structure, or a NULL pointer on error. *SRW_errno* is not set if an error occurs.

The event data structure is defined as follows:

```
typedef struct {
    SrEventTypeT evtype;
    union {
        SrTagT tag;
        StringT cdata;
        StringT entname;
        StringT pi;
    } u;
} SrEventT;
```

Examples

The following code gets the element GI from the event associated with the current table conversion object, and tests for a specific gi:

```
SrErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    SrEventT *tmpEventp;
    SrConvObjT tableObj;

    if ((eventp->evtype) == SR_EVT_BEGIN_CELL) {
        /* Get the current SR_OBJ_TABLE conversion object. */
        tblObj = Sr_GetCurConvObjOfType(SR_OBJ_TABLE);
        /* Get the corresponding event. */
        tempEventp = Sr_GetAssociatedEvent(tblObj);
        if(tmpEventp) {
            if (F_StrIEqual(tempEventp->u.tag.gi,
                           (StringT)"FINANCIAL_TABLE")) {
                /*Process the table in some way*/
            }
        }
    }
    . . .
}
```

See also

- [“Sw_GetAssociatedEvent\(\)” on page 288](#)
- [“SrConvObjT” on page 415](#)
- [“SrEventT” on page 416](#)

Sr_GetAttrVal()

Retrieves the attribute-value data structure for a single, specified attribute that is associated with the specified conversion object.

Synopsis

```
#include "fm_struct.h"

. . .
F_AttributeT Sr_GetAttrVal(SrConvObjT srObj,
    StringT fmAttrName);
```

Arguments

<i>srObj</i>	Conversion object to query
<i>fmAttrName</i>	Name of the attribute to query

Details

This function retrieves the proposed value for a specified attribute. It can only return a value if:

- The specified conversion object is associated with an event of type `SR_EVT_BEGIN_ELEM`.
- The corresponding element specifies a value for the named attribute with the element's start tag.
- The import template's EDD includes an appropriate attribute definition for the proposed FrameMaker element.

By default, an attribute in markup translates to a FrameMaker attribute of the same name. However, the current read/write rules can map the imported attribute to a FrameMaker attribute with a different name. The attribute name you provide must match the name of the FrameMaker attribute. For more information about how FrameMaker translates attributes, see the *FrameMaker Structure Application Developer's Guide*.

This function returns a structure of type `F_AttributeT`, which is defined as:

```
typedef struct (
    StringT name;
    F_StringsT values;
    UByteT valflags; /* validation error flag */
    UByteT allow; /* allow validation error as special case */
} F_AttributeT;
```

Note that `values` is a list of strings. Markup attribute values can be character data, or they can be tokens or lists of tokens. If the value is character data or a single token, then there is only one element in the list of strings (`values.len == 1`, indicating a single string in

`values.val[0]`). If the value is a list of tokens, then each string in the list corresponds to a single value token.

If you call `Sr_GetAttrVal()` before `srObj` is converted into a `FrameMaker` element, the function returns the attribute value currently proposed for the element. If you call `Sr_GetAttrVal()` after `srObj` is converted, the function returns the attribute value that was imported.

Important: When you no longer need the attribute-value data structure returned by this function, call `F_ApiDeallocateAttribute()` to delete the structure and free the memory allocated for it.

To get a list of attribute-values associated with a conversion object, call `Sr_GetAttrVals()`.

Returns

The requested attribute-value data structure, or a zeroed-out data structure on error. On error, `SRW_errno` is set to the following error codes:

- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_NO_SUCH_ATTR`

Examples

The following code retrieves the attribute-value data structure for an `ACCESS` attribute associated with the current conversion object. It then checks the list of value tokens and if both the `TOP_SECRET` and the `INTERNAL` token are present it drops that element. Note that `ACCESS` is the *case sensitive* name of the proposed *FrameMaker* attribute; with SGML you must ensure proper case for the attribute name. Incorrect case for the name is an error that causes `Sr_GetAttrVal()` to return an empty structure.

```
. . .
SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    F_AttributeT curAttr;
    IntT flags, errorStatus;
    UIntT i;
    BoolT topSecret, secret, internal, general;
    topSecret = secret = internal = general = False;
```



```
if (eventp->evtype == SR_EVT_BEGIN_ELEM) {
    curAttr = Sr_GetAttrVal(srObj, (StringT) "ACCESS");
    for (i=0; i < curAttr.values.len; i++) {
        if (F_StrIEqual((StringT)"TOP_SECRET",
                        curAttr.values.val[i]))
            topSecret = True;
        if (F_StrIEqual((StringT)"SECRET", curAttr.values.val[i]))
            secret = True;
        if (F_StrIEqual((StringT)"INTERNAL", curAttr.values.val[i]))
            internal = True;
        if (F_StrIEqual((StringT)"GENERAL", curAttr.values.val[i]))
            general = True;
    }
    /* When it is no longer needed, free the attribute. */
    F_ApiDeallocateAttribute(&curAttr);
    if (topSecret && internal) {
        /*Drop this element completely */
        flags = (SRW_DROP_ELEMENT_CONTENT | SRW_UNWRAP_ELEMENT);
        errorStatus = Sr_SetProcessingFlags(srObj, flags);
    }
    . . .
}
```

See also

- [“Sr_SetAttrVal\(\)” on page 184](#)
- [“Sr_GetAttrVals\(\)” on page 114](#)
- [“Structured_DeallocateAttrVal\(\)” on page 55](#)
- [“F_AttributeT” on page 410](#)
- [“SrConvObjT” on page 415](#)
- [“Primitive data types” on page 383](#) for more information on StringT

Sr_GetAttrVals()

Retrieves the list of attribute values for the specified conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

F_AttributesT Sr_GetAttrVals(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

This function retrieves the list of proposed values for a specified conversion object. The object can only have attribute values if:

- The specified conversion object is associated with an event of type `SR_EVT_BEGIN_ELEM`.
- The corresponding element in markup specifies at least one attribute, and includes a value for the attribute with the element's start tag.
- The import template's EDD includes appropriate attribute definitions for the proposed FrameMaker element.

By default, an attribute in markup translates to a FrameMaker attribute of the same name. However, the current read/write rules can map the imported attribute to a FrameMaker attribute with a different name. The attribute name you provide must match the name of the FrameMaker attribute. For more information about how FrameMaker translates attributes, see *FrameMaker Structure Application Developer's Guide*.

This function returns a structure of type `F_AttributesT`, which is defined as:

```
typedef struct {
    UIntT len; /* number of attributes */
    F_AttributeT *val; /* attributes array pointer */
} F_AttributesT;
```

val points to an array of attribute values, each one represented by a structure of type `F_AttributeT`, which is defined as:

```
typedef struct (
    StringT name;
    F_StringsT values;
    UByteT valflags; /* validation error flag */
    UByteT allow; /* allow validation error as special case */
} F_AttributeT;
```

For more information about single attribute values, see [“Sr_GetAttrVal\(\)” on page 111](#).

If you call `Sr_GetAttrVals()` before `srObj` is converted into a `FrameMaker` element, the function returns the proposed list of attribute values. If you call `Sr_GetAttrVals()` after `srObj` is converted, the function returns the list that was imported.

Important: When you no longer need the attribute-value data structure returned by this function, call `F_ApiDeallocateAttributes()` to delete the structure and free the memory allocated for it.

Returns

A list of attribute-value pairs, or a zeroed-out data structure on error. Not that a null structure can also indicate that the current object indicates an element that has no attributes defined for it. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The event handler prints out the name and value of each attribute in each element. It uses two nested loops to look at all the tokens that make up the values for each attribute in the list.

```
. . .
SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    F_AttributesT curAttrVals;
    StringT curAttrName, curValToken;
    UIntT i, j;

    if (eventp->evtype == SR_EVT_BEGIN_ELEM) {
        curAttrVals = Sr_GetAttrVals(srObj);
        for(i=0; i< curAttrVals.len; i++) {
            for (j=0; j< curAttrVals.val[i].values.len; j++) {
                F_Printf(NULL, "\nValue of %s is: %s",
                    curAttrVals.val[i].name,
                    curAttrVals.val[i].values.val[j]);
            }
        }
        /* When no longer needed, free the attributes. */
        F_ApiDeallocateAttributes(&curAttrVals);
    }
    . . .
}
```

See also

- [“Sr_SetAttrVals\(\)” on page 186](#)
- [“Sr_GetAttrVal\(\)” on page 111](#)

- [“Structured_DeallocateAttrVals\(\)” on page 56](#)
- [“F_AttributesT” on page 410](#)
- [“SrConvObjT” on page 415](#)
- [“SRW_errno” on page 418](#)

Sr_GetBookCompFilePath()

Retrieves a pointer to the filepath for importing a file in a book for the specified `SR_OBJ_BOOK_COMP` conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
FilePathT *Sr_GetBookCompFilePath(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

To get a pointer to the filepath for a book component associated with an `SR_OBJ_BOOK_COMP` conversion object, call `Sr_GetBookCompFilePath()`. A book component is a FrameMaker document that is part of a book file. To get the filepath for a book file, call `Sr_GetBookFilePath()` instead.

If you call `Sr_GetBookCompFilePath()` before *srObj* is converted, the function returns the filepath proposed for creating the book component file.

If you call `Sr_GetBookCompFilePath()` after *srObj* is converted, the function returns the filepath that was used to create the corresponding book component file.

`FilePathT` is a pointer to a platform-independent representation of a platform-specific path. For more information about working with platform-independent representations of filepaths, see the *FDK Programmer's Reference*. Use `F_FilePathFree()` to free the filepath when you are done with it.

Returns

A pointer to the platform-independent filepath, or `NULL` on error. On error, `SRW_errno` is set to one of the following possible codes:

- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_NOT_BOOK_COMP`

Examples

The following code returns a pointer to the book component path for the current conversion object if it is of type `SR_OBJ_BOOK_COMP`:

```
. . .  
FilePathT *bookCompFp;  
. . .  
if(Sr_GetObjType(srObj) == SR_OBJ_BOOK_COMP)  
    bookCompFp = Sr_GetBookCompFilePath(srObj);  
. . .
```

See also

- [“Sr_SetBookCompFilePath\(\)” on page 189](#)
- [“Sr_GetBookFilePath\(\)” on page 117](#)
- [“SrConvObjT” on page 415](#)
- [“FilePathT” on page 409](#)
- [“SRW_erno” on page 418](#)

Sr_GetBookFilePath()

Gets a pointer to the filepath of a book file for a specified conversion object of type `SR_OBJ_BOOK`.

Synopsis

```
#include "fm_struct.h"  
. . .  
FilePathT *Sr_GetBookFilePath (SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

`Sr_GetBookFilePath()` gets a pointer to a book file’s location for a specified conversion object of type `SR_OBJ_BOOK`. To retrieve the filepath for individual FrameMaker documents within a book file, call `Sr_GetBookCompFilePath()` instead.

If you call `Sr_GetBookFilePath()` before *srObj* is converted, the function returns the filepath proposed for creating the book file. If you call `Sr_GetBookFilePath()` after *srObj* is converted, the function returns the filepath that was used to create the book file.

`FilePathT` is a pointer to a platform-independent representation of a platform-specific path. For more information about working with platform-independent representations of filepaths, see the *FDK Programmer’s Reference*. Use `F_FilePathFree()` to free the returned structure when you are done with it.

Returns

A pointer to the filepath of a book file, or NULL for an error. On error, SRW_errno is set to SRW_E_WRONG_OBJ_TYPE.

Examples

The following code retrieves a pointer to the filepath of a book file. If the pointer is NULL, it prompts for a valid filepath, and sets it; the code assumes your code defines the function, PromptForBookFilePath().

```
. . .
FilePathT *bookFilePath;
StringT fileName;
. . .
if (Sr_GetObjType(srObj) == SR_OBJ_BOOK)
{
    if((bookFilePath = Sr_GetBookFilePath(srObj)) == NULL)
    {
        PromptForBookFilePath(fileName, pform);
        bookFilePath = F_PathNameToFilePath(
            fileName, NULL, FDefaultPath);
    }
    Sr_SetBookFilePath(srObj, bookFilePath);
    F_FilePathFree(bookFilePath);
}
. . .
```

See also

- [“Sr_SetBookFilePath\(\)” on page 190](#)
- [“Sr_GetBookCompFilePath\(\)” on page 116](#)
- [“Srw_GetStructuredDocFilePath\(\)” on page 271](#)
- [“SrConvObjT” on page 415](#)
- [“FilePathT” on page 409](#)
- [“SRW_errno” on page 418](#)

Sr_GetBookId()

Retrieves the ID for the FrameMaker book file that is associated with the specified import conversion object. Note that the conversion object must be one of type `SR_OBJ_BOOK`.

Synopsis

```
#include "fm_struct.h"

. . .

F_ObjHandleT Sr_GetBookId(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

This function retrieves the ID of a FrameMaker book file. If you have the book ID, you can then get the ID of any document in the book.

The ID doesn't exist until the book is converted. This means an event of type `SR_EVT_BEGIN_BOOK` must have occurred, and the corresponding conversion object must have been converted. If you call `Sr_GetBookId()` before an `SR_OBJ_BOOK` is converted, this function returns `NULL`.

This function returns the ID of the book that is associated with the specified conversion object. You can change the book ID directly in a custom event handler with `Sr_SetBookId()`.

Returns

A FrameMaker book ID, or `NULL`. `NULL` may indicate that the book was not yet created, or may indicate another error condition. On error, `SRW_errno` is set to `SRW_E_INVALID_CONV_OBJ`, `SRW_E_BAD_OBJ_HANDLE`, or `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code traps a book import event, and then retrieves the associated book ID:

```
SrErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj, parentObj = 0;
{
    F_ObjHandleT bookId;
```

```
if ((eventp->evtype) == SR_EVT_BEGIN_BOOK) {
    if (Sr_GetObjType(srObj) == SR_OBJ_BOOK) {
        bookId = Sr_GetBookId(srObj);
        /*Process the book in some way*/
        . . .
    }
    . . .
}
. . .
}
```

See also

- [“Sr_SetBookId\(\)” on page 192](#)
- [“Sr_GetDocId\(\)” on page 132](#)
- [“Sr_GetFlowId\(\)” on page 135](#)
- [“Sr_GetFmElemId\(\)” on page 138](#)
- [“SrConvObjT” on page 415](#)
- [“SRW_erno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `F_ObjHandleT`

Sr_GetCharFmt()

Retrieves the character format tag associated with the specified conversion object of type `SR_OBJ_SPECIAL_CHAR`.

Synopsis

```
#include "fm_struct.h"
. . .
StringT Sr_GetCharFmt(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

Call `Sr_GetCharFmt()` to retrieve the name of the character format tag associated with a FrameMaker `SR_OBJ_SPECIAL_CHAR` conversion object. A character format tag is the name of a character format as it appears in the FrameMaker Character Catalog. It indicates the character format that is used to control the appearance of the object in a FrameMaker document.

FrameMaker generates a `SR_OBJ_SPECIAL_CHAR` conversion object when importing an `SDATA` entity that has a corresponding read/write rule to convert that entity to a special FrameMaker character. There are specific issues associated with special characters. For example, the character format for any special character must specify a non-text font such as Symbol. (For more information see the online manual, *FrameMaker Structure Application Developer's Guide*.)

If you call this function before an object is converted, it reports the proposed character format tag for the object.

If you call this function after an object is converted, it reports the character format tag of the FrameMaker element that was created by the conversion. To set the character format tag to use with the special character conversion object before it is converted, call `Sr_SetCharFmt()`.

Returns

Name of the character format tag, or `NULL` on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code prints out the character tag associated to each special character object:

```
. . .
StringT s;
. . .
if(Sr_GetObjType(srObj) == SR_OBJ_SPECIAL_CHAR) {
    s = Sr_GetCharFmt(srObj);
    if(!F_StrIsEmpty(s)) {
        F_Printf(NULL, "\nSpecial Char is tagged: %s", s);
        F_ApiDeallocateString(&s);
    }
}
. . .
```

See also

- [“Sr_SetCharFmt\(\)” on page 194](#)
- [“Sr_GetFmChar\(\)” on page 137](#)
- [“SrConvObjT” on page 415](#)
- [“SRW_errno” on page 418](#)

Sr_GetChildConvObj()

Returns a conversion object that is immediately subordinate to the specified import object.

Synopsis

```
#include "fm_struct.h"

. . .

SrConvObjT Sr_GetChildConvObj(SrConvObjT parentObj);
```

Arguments

parentObj Parent conversion object to query

Details

As import events occur, corresponding conversion objects are created. These conversion object/event pairs are pushed onto a stack that represents the hierarchy of currently open conversion objects. Note that for container objects, import events come in pairs; for every BEGIN event to open a container there is a corresponding END event to close it. A conversion object is opened when the BEGIN event occurs, and is closed when the END event occurs.

Note that any number of nested container objects can be open at one time. However, among sibling objects only one sibling can be open at a time, because an object is removed from the stack as soon as it is closed.

```
<LEVEL_ONE>
  <LEVEL_TWO>
    <LEVEL_THREE>
      <LEVEL_FOUR>
        <LEVEL_FIVE></LEVEL_FIVE> (to be removed from stack)
```

Notice that LEVEL_FIVE has just posted it's closing object, and will be removed from the stack. The next conversion object could easily be another LEVEL_FIVE object, but there will never be two open LEVEL_FIVE objects on the stack at the same time.

Sr_GetChildConvObj() returns the ID of the the open conversion object that is a child of the specified object. For example, starting from the current object of type *SR_OBJ_DOC*, you can use this function to get the highest level element in the document.

Returns

A conversion object, or *NULL* under the following conditions:

- No child object is associated with the specified conversion object.
- Child objects are associated with the specified conversion object, but no child object is open.

Examples

The following code uses the current document as the entry point, and lists the ancestry of a table element.

```
#include "fm_struct.h"

SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    SrConvObjT elemObj;
    SrEventT *elemEvtp;

    if(F_StrIEqual(eventp->u.tag.gi, "TABLE")) {
        F_Printf(NULL, "\n\nAncestry of %s is:",
                eventp->u.tag.gi);
        elemObj = Sr_GetCurConvObjOfType(SR_OBJ_DOC);
        while (elemObj) {
            elemEvtp = Sr_GetAssociatedEvent(elemObj);
            if(elemEvtp->evtype == SR_EVT_BEGIN_ELEM)
                F_Printf(NULL, "\n    %s", elemEvtp->u.tag.gi);
            elemObj = Sr_GetChildConvObj(elemObj);
        }
    }
    return (Sr_Convert(eventp, srObj));
}
```

See also

- [“Sr_GetCurConvObj\(\)” on page 128](#)
- [“Sr_GetParentConvObj\(\)” on page 153](#)
- [“SrConvObjT” on page 415](#)

Sr_GetColSpecs()

Retrieves a list of *CALS* column specifications (*colspecs*) for a table or table part (heading, body, or footing) associated with the specified import conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

SrwColSpecsT Sr_GetColSpecs(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

To retrieve the list of *colspecs* for a table or table part associated with an import conversion object, call *Sr_GetColSpecs()*. Use this function only for *CALS* tables.

srObj must be one of the following types:

- *SR_OBJ_TABLE*
- *SR_OBJ_TABLE_HEADING*
- *SR_OBJ_TABLE_BODY*
- *SR_OBJ_TABLE_FOOTING*

Note that in a strict *CALS* model, calling *Sr_GetColSpecs()* with an object of type *SR_OBJ_TABLE_BODY* always returns an empty data structure.

Important: Ordinarily, *Sr_GetColSpecs()* is not called for the current conversion object, but for its parent object. For example, if the current object is of type *SR_OBJ_TABLE*, then the object's *colspecs* have not yet been set. On the other hand, when the current object is of type *SR_OBJ_TABLE_ROW* or *SR_OBJ_TABLE_CELL*, there might be good reason to call *Sr_GetColSpecs()* on the parent table object to determine column information for the row or cell.

You can set or change the list of *colspecs* associated with a conversion object by calling *Sr_SetColSpecs()*.

After it is no longer needed, the *SrwColSpecsT* structure returned by this function should be deallocated with *Srw_DeallocateColSpecs()*.

Returns

An *SrwColSpecsT* structure, or a zeroed-out structure on error. On error, *SRW_errno* is set to *SRW_E_WRONG_OBJ_TYPE*.

Examples

The following code demonstrates a call to `Sr_GetColSpecs()`:

```
. . .
SrwErrorT DetermineColumnInfo(SrConvObjT srObj)
{
    SrwColSpecsT colspecs;
    SrConvObjT tblObj;
    switch(Sr_GetObjType(srObj))
    {
        case SR_OBJ_TABLE_ROW:
            /* Retrieve the current table object. */
            tblObj = Sr_GetCurConvObjOfType(SR_OBJ_TABLE);
            /* Retrieve copy of colspecs for table object. */
            colspecs = Sr_GetColSpecs(tblObj);
            . . .
            /* When no longer needed, free colspecs. */
            Srw_DeallocateColSpecs(&colspecs);
            . . .
    }
}
```

See also

- [“Sr_SetColSpecs\(\)” on page 195](#)
- [“Srw_DeallocateColSpecs\(\)” on page 251](#)
- [“SrConvObjT” on page 415](#)
- [“SrwColSpecsT” on page 419](#)
- [“SRW_errno” on page 418](#)

Sr_GetCurColSpecByColNum()

Retrieves a specified *CALS* column specification (*colspec*) by column number from the stack of the currently open table or table-part conversion objects.

Synopsis

```
#include "fm_struct.h"
. . .
SrwColSpecT Sr_GetCurColSpecByColNum(IntT colnum);
```

Arguments

colnum Column number of *colspec* for which to search

Details

If you know the column number of a *colspec* associated with a table or table-part conversion object on the conversion object stack, you can call *Sr_GetCurColSpecByColNum()* to retrieve it. Use this function only with *CALS* tables. The first *colnum* in a table is 0.

You can also retrieve a *colspec* by name with *Sr_GetCurColSpecByName()*.

After it is no longer needed, you should delete the *colspec* structure returned by this function, and free the memory allocated for it with a call to *Srw_DeallocateColSpec()*.

Returns

A *SrwColSpecT* structure. If a matching *colspec* is not found, a zeroed-out structure is returned, and *SRW_errno* is set to *SRW_E_FAILURE*.

Examples

The following code retrieves the second *colspec* (*colnum* = 1) in a table from the stack of conversion objects, and later deallocates the *colspec* structure returned by *Sr_GetCurColSpecByColNum()*:

```
. . .
SrwColSpecT numberedColSpec;
. . .
numberedColSpec = Sr_GetCurColSpecByColNum(1);
. . .
Srw_DeallocateColSpec(&numberedColSpec);
. . .
```

See also

- [“Sr_GetCurColSpecByName\(\)” on page 127](#)
- [“Srw_DeallocateColSpec\(\)” on page 250](#)
- [“SrwColSpecT” on page 418](#)
- [“SRW_errno” on page 418](#)

- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `IntT`

Sr_GetCurColSpecByName()

Retrieves a specified *CALS* column specification (*colspec*) by column name from the stack of currently open table or table-part conversion objects.

Synopsis

```
#include "fm_struct.h"

. . .

SrwColSpecT Sr_GetCurColSpecByName(StringT name);
```

Arguments

name Name of *colspec* for which to search

Details

If you know the name of a *colspec* associated with a table or table-part conversion object, you can call `Sr_GetCurColSpecByName()` to retrieve the *colspec*. Use this function only with *CALS* tables.

You can also retrieve a *colspec* by number with `Sr_GetCurColSpecByColNum()`.

Use `Srw_DeallocateColSpec()` to free the structure when you are done with it.

Returns

A `SrwColSpecT` structure. If a matching *colspec* is not found, a zeroed-out structure is returned, and `SRW_errno` is set to `SRW_E_FAILURE`.

Examples

The following code retrieves a *colspec* from the stack of conversion objects, and later deallocates the *colspec* structure returned by `Sr_GetCurColSpecByName()`:

```
. . .
SrwColSpecT namedColSpec;
. . .
namedColSpec = Sr_GetCurColSpecByName((StringT) "Definition");
. . .
Srw_DeallocateColSpec(&namedColSpec);
. . .
```

See also

- [“Sr_GetCurColSpecByColNum\(\)” on page 126](#)
- [“Srw_DeallocateColSpec\(\)” on page 250](#)
- [“SrwColSpecT” on page 418](#)

- [“SRW_erno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_GetCurConvObj()

Retrieves the conversion object most recently passed to the import event handler.

Synopsis

```
#include "fm_struct.h"
. . .
SrConvObjT Sr_GetCurConvObj(VoidT);
```

Arguments

None.

Details

The current object is the one associated with the import event that is currently being processed by the import event handler. The current conversion object is a primary entry into the conversion object stack.

Because event handlers are passed the current object as a parameter, you would generally reserve this function reserved for use by procedures that don't get the current object as a parameter.

Given a conversion object, you can get information about it; among other things, about its type, its associated event, its parent object, and the Generic Identifier for the previous sibling conversion object (if one exists).

Returns

The current conversion object.

Examples

The following code might be found in a function that wasn't passed the current object or the current event. It retrieves the current conversion object from the stack, then gets the object type, and if the object is an element, it gets the associated GI:

```
. . .
SrConvObjT srObj;
SrEventT *eventp;
SrObjTypeT srObjType;
. . .
```



```
srObj = Sr_GetCurConvObj();
srObjType = Sr_GetObjType(srObj);
if (srObjType == SR_OBJ_ELEM) {
    eventp = Sr_GetAssociatedEvent(srObj);
    if(F_StrIEqual(eventp->u.tag.gi, (StringT)"body") {
        /*Now you have the element tag associated with */
        /*the current event... Process accordingly. */
        . . .
    }
    . . .
}
```

See also

- [“Sr_GetChildConvObj\(\)” on page 122](#)
- [“Sr_GetParentConvObj\(\)” on page 153](#)
- [“Sr_EventHandler\(\)” on page 107](#)
- [“SrConvObjT” on page 415](#)

Sr_GetCurConvObjOfType()

Retrieves the most recently opened import conversion object of a specific type.

Synopsis

```
#include "fm_struct.h"
. . .
SrConvObjT Sr_GetCurConvObjOfType(SrObjTypeT objType);
```

Arguments

objType Type of conversion object for which to search

Details

This function checks the conversion object stack, starting from the current object, for the nearest open conversion object of the specified type. Consider the following example:

```
<SESSION>
  <DOC>
    <ELEM (Section)>
      <ELEM (Para)>
        <XREF>
```

The current conversion object is the XREF. Two elements are open; one for the section and one for a paragraph in the section. The current object of type ELEM is the Para element because it is the nearest open object of that type.

A common use for this function is to retrieve the conversion object for the FrameMaker document into which you are importing markup. For example, you might need to get the

document's ID, available in the document's conversion object, because you need to pass it as an argument to an FDK function or to another structure import/export API function.

Returns

The conversion object, or `NULL` if there is no open object that matches the specified type. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code retrieves the current document conversion object from the stack. It then stores the document's ID in a variable that persists across multiple events. Of course, the call to get the document conversion object must occur after the corresponding `SR_EVT_BEGIN_DOC` event has been converted. Note that the code does not overwrite `srObj`; the handler requires `srObj` to call `Sr_Convert()`.

```
. . .
F_ObjHandleT docId;
. . .
SrwErrorT Sr_EventHandler(eventp, srObj)
    SrEventT *eventp;
    SrConvObjT srObj;
{
    SrConvObjT docObj;
    . . .
    if (docObj = Sr_GetCurConvObjOfType(SR_OBJ_DOC))
        docId = Sr_GetDocId(docObj);
    . . .
```

See also

- [“Sr_GetCurConvObj\(\)” on page 128](#)
- [“Sr_GetChildConvObj\(\)” on page 122](#)
- [“Sr_GetParentConvObj\(\)” on page 153](#)
- [“SrConvObjT” on page 415](#)
- [“SrObjTypeT” on page 395](#)
- [“SRW_errno” on page 418](#)

Sr_GetCurSpanSpecByName()

Retrieves a specified *CALS* span specification (*spanspec*) by span name from the stack of currently open table or table-part conversion objects.

Synopsis

```
#include "fm_struct.h"
. . .
SrwSpanSpecT Sr_GetCurSpanSpecByName(StringT name);
```

Arguments

name Name of *spanspec* for which to search

Details

If you know the name of a *spanspec* associated with a table or table-part conversion object on the conversion object stack, you can call `Sr_GetCurSpanSpecByName()` to retrieve it.

Returns

A `SrwSpanSpecT` structure. If a matching *spanspec* is not found, a zeroed-out structure is returned, and `SRW_errno` is set to `SRW_E_FAILURE`. Use `Srw_DeallocateSpanSpec()` to free the returned structure when you are done with it.

Examples

The following code retrieves a *spanspec* from the stack of conversion objects, and later deallocates the *spanspec* structure returned by `Sr_GetCurSpanSpecByName()`:

```
. . .
SrwSpanSpecT namedSpanSpec;
. . .
namedSpanSpec = Sr_GetCurSpanSpecByName((StringT) "Definition");
. . .
Srw_DeallocateSpanSpec(&namedSpanSpec);
. . .
```

See also

- [“Srw_DeallocateSpanSpec\(\)” on page 254](#)
- [“SrwSpanSpecT” on page 422](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_GetDocId()

Retrieves the ID for the FrameMaker document file that is associated with the specified import conversion object. Note that the conversion object must be one of type `SR_OBJ_DOC`.

Synopsis

```
#include "fm_struct.h"

. . .

F_ObjHandleT Sr_GetDocId(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

This function retrieves the ID of a FrameMaker document.

Some FDK and structure import/export API functions require a FrameMaker document ID as an argument. For example, `F_ApiImport()`, which can be called to write directly to a FrameMaker document, expects an ID argument.

The ID doesn't exist until the document is converted. This means an event of type `SR_EVT_BEGIN_DOC` must have occurred, and the corresponding `SR_OBJ_DOC` or `SR_OBJ_BOOK_COMP` conversion object must have been converted. If you call `Sr_GetDocId()` before one of these objects is converted, this function returns 0.

This function returns the ID of the document that is associated with the specified conversion object. You can change the document ID directly in a custom event handler with `Sr_SetDocId()`.

Returns

A FrameMaker document ID, or 0. 0 may indicate that the object is not yet converted, or may indicate another error condition. On error, `SRW_errno` is set to `SRW_E_INVALID_CONV_OBJ` or `SRW_E_BAD_OBJ_HANDLE`.

Examples

The following code traps a document import event, and then retrieves the associated document ID:

```
. . .

SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    F_ObjHandleT docId;
```

```
    if ((eventp->evtype) == SR_EVT_BEGIN_DOC) {
        Sr_Convert(eventp, srObj);
        if (Sr_GetObjType(srObj) == SR_OBJ_DOC) {
            docId = Sr_GetDocId(srObj);
            /*Process the document in some way*/
            . . .
        }
        return(SRW_E_SUCCESS);
    }
    . . .
}
```

See also

- [“Sr_SetDocId\(\)” on page 197](#)
- [“Sr_GetBookId\(\)” on page 119](#)
- [“Sr_GetFlowId\(\)” on page 135](#)
- [“Sr_GetFmElemId\(\)” on page 138](#)
- [“Sr_GetFmObjId\(\)” on page 141](#)
- [“SrConvObjT” on page 415](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `F_ObjHandleT`

Sr_GetExtEntityFilePath()

Retrieves the filepath for the external entity associated with the current conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

FilePathT *Sr_GetExtEntityFilePath(StringT entname);
```

Arguments

entname The name of the external entity

Details

To determine the filepath of the external entity associated with the current conversion object, call `Sr_GetExtEntityFilePath()`. `entname` should be set to the name of the entity whose filepath you want to return.

Important: This function is to be used with import event handlers. You can use it with an entity handler, but there is no need; the default file path for an entity is one of the arguments passed to the entity handler. `Sr_GetExtEntityFilePath()` effectively duplicates that information if you use it within the entity handler. For more information, see [“Srw_EntityHandler\(\)” on page 259](#).

`FilePathT` is a pointer to a platform-independent representation of a platform-specific path. For more information about working with platform-independent representations of filepaths, see the *FDK Programmer's Reference*. Use `F_FilePathFree()` to free the returned filepath when you are done with it.

Returns

A pointer to the platform-independent filepath for the external entity, or `NULL` on error. On error, `SRW_errno` is set to `SRW_E_BAD_VALUE`.

Examples

The following code gets the filepath for an entity associated with a conversion object and prints the path to the console:

```
. . .

SrwErrorT Sr_EventHandler(eventp, srObj)
    SrEventT *eventp;
    SrConvObjT srObj;
{
    switch(eventp->evtype) {

        FilePathT *entFilePath;
        StringT s;
```

```
case SR_EVT_BEGIN_ENTITY:
    if(entFilePath =
        Sr_GetExtEntityFilePath(eventp->u.entname))
    {
        s = F_FilePathToPathName(entFilePath, FDefaultPath);
        F_Printf(NULL, "\nEntity path is: %s", s);
        F_ApiDeallocateString(&s);
        F_FilePathFree(entFilePath);
    }
    break;
}
return(Sr_Convert(eventp, srObj));
}
```

. . .

See also

- [“Sr_GetTextInsetFilePath\(\)” on page 174](#)
- [“Sr_GetBookCompFilePath\(\)” on page 116](#)
- [“Sr_GetBookFilePath\(\)” on page 117](#)
- [“FilePathT” on page 409](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_GetFlowId()

Retrieves the ID of the main flow for the FrameMaker document file that is associated with the specified import conversion object. Note that the conversion object must be one of type `SR_OBJ_DOC`.

Synopsis

```
#include "fm_struct.h"

. . .
F_ObjHandleT Sr_GetFlowId(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

This function retrieves the ID of the main text flow in a FrameMaker document. This text flow will contain the stream of imported document content. Usually, the main flow is the

default flow for the current language (in the English language, flow 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.

If you have the main flow ID, you can then get the ID of any content object in the flow. For example, given a flow ID, you can then get the ID of the highest level structure element in that flow.

The flow ID doesn't exist until the document is converted. This means an event of type `SR_EVT_BEGIN_DOC` must have occurred, and the corresponding conversion object must have been converted. If you call `Sr_GetFlowId()` before an `SR_OBJ_DOC` is converted, this function returns 0.

This function returns the ID of the flow associated with the specified conversion object. You can change the flow ID directly in a custom event handler with `Sr_SetFlowId()`.

Returns

A FrameMaker flow ID, or 0. 0 may indicate that the object is not yet converted, or may indicate another error condition. On error, `SRW_errno` is set to `SRW_E_INVALID_CONV_OBJ` or `SRW_E_BAD_OBJ_HANDLE`.

Examples

The following code traps a document import event, and then retrieves the associated flow ID:

```
. . .
SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    F_ObjHandleT flowId;

    if ((eventp->evtype) == SR_EVT_BEGIN_DOC) {
        Sr_Convert(eventp, srObj);
        if (Sr_GetObjType(srObj) == SR_OBJ_DOC) {
            flowId = Sr_GetFlowId(srObj);
            /* Use the document flow in some process. */
        }
    }
    . . .
```

See also

- [“Sr_SetFlowId\(\)” on page 199](#)
- [“Sr_GetBookId\(\)” on page 119](#)
- [“Sr_GetDocId\(\)” on page 132](#)
- [“Sr_GetFmElemId\(\)” on page 138](#)
- [“Sr_GetFmObjId\(\)” on page 141](#)

- [“SrConvObjT” on page 415](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `F_ObjHandleT`

Sr_GetFmChar()

Retrieves the FrameMaker special characters associated with the specified import conversion object of type `SR_OBJ_SPECIAL_CHAR`.

Synopsis

```
#include "fm_struct.h"
. . .
UCharT Sr_GetFmChar(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

You use read/write rules to map an *SDATA* entity to a special character in FrameMaker, or you can specify the mapping directly in the entity declaration. On import, FrameMaker substitutes the entity with the specified character. This function returns that character.

If you call this function before an object is converted, it returns the special character mapping proposed for the object in FrameMaker. If you call this function after an object is converted, it returns the special character mapping that was assigned to the object in FrameMaker.

You can assign special character mappings with `Sr_SetFmChar()`.

Returns

A special character, or 0. 0 may indicate that the object is not yet converted, or it may indicate another error condition. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code retrieves the special character associated with the current conversion object if it is of type `SR_OBJ_SPECIAL_CHAR`:

```
. . .
UCharT specChar;
. . .
if(Sr_GetObjType(srObj) == SR_OBJ_SPECIAL_CHAR)
    specChar = Sr_GetFmChar(srObj);
. . .
```

See also

- [“Sr_SetFmChar\(\)” on page 202](#)
- [“Sr_GetCharFmt\(\)” on page 120](#)
- [“SrConvObjT” on page 415](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `UCharT`

Sr_GetFmElemId()

Retrieves the ID of the FrameMaker structure element that is associated with the specified import conversion object. Note that the conversion object must be one of a type that is specified below.

Synopsis

```
#include "fm_struct.h"
. . .
F_ObjHandleT Sr_GetFmElemId(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

This function retrieves the ID of a FrameMaker element. The element is associated with a specified conversion object of one of the following types:

SR_OBJ_ELEM	SR_OBJ_TABLE	SR_OBJ_TABLE_TITLE
SR_OBJ_TABLE_HEADING	SR_OBJ_TABLE_BODY	SR_OBJ_TABLE_FOOTING
SR_OBJ_TABLE_ROW	SR_OBJ_TABLE_CELL	SR_OBJ_TABLE_COLSPEC
SR_OBJ_TABLE_SPANSPEC	SR_OBJ_GRAPHIC	SR_OBJ_EQUATION
SR_OBJ_VARIABLE	SR_OBJ_MARKER	SR_OBJ_XREF
SR_OBJ_FOOTNOTE		

The element ID doesn't exist until the element is converted. This means an event of type `SR_EVT_BEGIN_ELEM` must have occurred, and the corresponding conversion object must have been converted. If you call `Sr_GetFmElemId()` before the specified conversion object is converted, this function returns 0.

Returns

The ID of a FrameMaker element, or 0. 0 may indicate that the object is not yet converted, or may indicate another error condition. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code traps an element import event and checks for a DOC element. If it is a DOC element, the code converts the element and then saves the element ID for later processing.

```
. . .
F_ObjHandleT docElemId;
. . .
SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    F_ObjHandleT docElemId;

    if ((eventp->evtype) == SR_EVT_BEGIN_ELEM) {
        if (F_StrIEqual(eventp->u.tag.gi, (StringT)"DOC")) {
            Sr_Convert(eventp, srObj);
            docElemId = Sr_GetFmElemId(srObj);
            . . .
            return(SRW_E_SUCCESS);
        }
    }
    . . .
}
```

See also

- [“Sr_UseFmElemId\(\)” on page 236](#)
- [“Sr_GetBookId\(\)” on page 119](#)
- [“Sr_GetDocId\(\)” on page 132](#)
- [“Sr_GetFlowId\(\)” on page 135](#)
- [“Sr_GetFmObjId\(\)” on page 141](#)
- [“SrConvObjT” on page 415](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `F_ObjHandleT`

Sr_GetFmElemTag()

Retrieves the tag for a FrameMaker element associated with the specified conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
StringT Sr_GetFmElemTag(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

This function retrieves the tag (name) of the FrameMaker element associated with the specified conversion object. You might want this information to pass to another function, or because you want to verify or modify tag naming conventions in your FrameMaker document. *srObj* must be a conversion object of the following types:

SR_OBJ_ELEM	SR_OBJ_TABLE	SR_OBJ_TABLE_TITLE
SR_OBJ_TABLE_HEADING	SR_OBJ_TABLE_BODY	SR_OBJ_TABLE_FOOTING
SR_OBJ_TABLE_ROW	SR_OBJ_TABLE_CELL	SR_OBJ_TABLE_COLSPEC
SR_OBJ_TABLE_SPANSPEC	SR_OBJ_GRAPHIC	SR_OBJ_EQUATION
SR_OBJ_VARIABLE	SR_OBJ_MARKER	SR_OBJ_XREF
SR_OBJ_FOOTNOTE		

If you call this function before the object is converted, it reports the tag proposed for the object in your FrameMaker document. If you call this function after an object is converted, it reports the tag that was assigned to the object in your document.

To specify the tag for a conversion object to use, call `Sr_SetFmElemTag()`.

Returns

The tag of a FrameMaker element, or NULL on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code prints out the tag for the proposed FrameMaker element, as well as the generic identifier for the imported element. While you may never need to get both strings, this example shows the difference between getting the two.

```
#include "fm_struct.h"
#include "fstrings.h"
```

```
SrErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
StringT s;
{
    if (eventp->evtype == SR_EVT_BEGIN_ELEM) {
        s = Sr_GetFmElemTag(srObj);
        F_Printf(NULL, "\\FrameMaker elem tag is: %s", s);
        F_ApiDeallocateString(&s);
        F_Printf(NULL, "Imported elem tag is: %s",
            eventp->u.tag.gi);
    }
    return(Sr_Convert(eventp, srObj));
}
```

See also

- [“Sr_SetFmElemTag\(\)” on page 203](#)
- [“Sr_GetFmElemId\(\)” on page 138](#)
- [“SrConvObjT” on page 415](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_GetFmObjId()

Retrieves the ID of a nonelement FrameMaker object that is associated with the specified import conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
F_ObjHandleT Sr_GetFmObjId(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

This function retrieves the ID of a nonelement FrameMaker object. Note that the FrameMaker user interface forces most document objects to be wrapped in structure elements. The only unwrapped objects a user can insert are of type `SR_OBJ_VARIABLE` or `SR_OBJ_MARKER`.

The object ID doesn't exist until the object is converted. This means an event of the following type must have occurred:

- SR_EVT_BEGIN_ELEM
- SR_EVT_BEGIN_ENTITY
- SR_EVT_PI
- SR_EVT_CDATA
- SR_EVT_RE

Assuming one of the above events, a corresponding conversion object of one of the following types must have been converted:

SR_OBJ_COLSPEC	SR_OBJ_SPANSPEC	SR_OBJ_TABLE_HEADING
SR_OBJ_ELEM	SR_OBJ_TABLE	SR_OBJ_TABLE_ROW
SR_OBJ_FOOTNOTE	SR_OBJ_TABLE	SR_OBJ_TABLE_TITLE
SR_OBJ_MARKER	SR_OBJ_TABLE_BODY	SR_OBJ_VARIABLE
SR_OBJ_RUBI	SR_OBJ_TABLE_CELL	SR_OBJ_XREF
SR_OBJ_RUBI_GROUP	SR_OBJ_TABLE_FOOTING	

If you call `Sr_GetFmElemId()` before the specified conversion object is converted, this function returns 0.

Returns

The ID of a FrameMaker object, or 0. 0 may indicate that the object is not yet converted, or it may indicate another error condition. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code traps an element import event and checks for a marker. If the element is a marker, it converts the element and saves the marker ID for later processing.

```
. . .
SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    F_ObjHandleT markerId;
```

```
    if ((eventp->evtype) == SR_EVT_BEGIN_ELEM) {
        if (Sr_GetObjType(srObj) == SR_OBJ_MARKER) {
            Sr_Convert(eventp, srObj);
            markerId = Sr_GetFmObjId(srObj);
        }
        . . .
    }
```

See also

- [“Sr_GetFmElemTag\(\)” on page 140](#)
- [“Sr_GetBookId\(\)” on page 119](#)
- [“Sr_GetDocId\(\)” on page 132](#)
- [“Sr_GetFlowId\(\)” on page 135](#)
- [“Sr_GetFmElemId\(\)” on page 138](#)
- [“SrConvObjT” on page 415](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `F_ObjHandleT`

Sr_GetFmText()

Retrieves FrameMaker text for the specified conversion object of type `SR_OBJ_TEXT`.

Synopsis

```
#include "fm_struct.h"
. . .
StringT Sr_GetFmText(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

FrameMaker imports CDATA and PCDATA via conversion objects of type `SR_OBJ_TEXT`. Conversion objects of this type are opened for every event of type `SR_EVT_CDATA`. Note that more than one such event can be posted for a single entity reference, but no more than one event is posted for a single character reference.

If you call this function before an object is converted, it supplies the text proposed for the object in FrameMaker. If you call this function after an object is converted, it supplies the text that was assigned to the object in FrameMaker.

To specify FrameMaker text for a conversion object of type `SR_OBJ_TEXT`, call `Sr_SetFmText()`.

Returns

A string containing the FrameMaker text associated with a conversion object, or `NULL` on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code traps `CDATA` or `PCDATA` and strips leading spaces out of the text associated with the conversion object:

```
#include "fm_struct.h"
#include "fstrings.h"

SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    StringT s, p;

    if (eventp->evtype == SR_EVT_CDATA) {
        s = Sr_GetFmText(srObj);
        if (!(F_StrIsEmpty(string))) {
            F_StrStripLeadingSpaces(s);
            Sr_SetFmText(srObj, s);
        }
        F_ApiDeallocateString(&s);
    }
    return(Sr_Convert(eventp, srObj));
}
```

See also

- [“Sr_SetFmText\(\)” on page 205](#)
- [“SrConvObjT” on page 415](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_GetImportFileHint()

Retrieves the file format and filter to use to import a graphic or equation associated with an import conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

StringT Sr_GetImportFileHint(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

Call `Sr_GetImportFileHint()` to retrieve the string identifying the file format and filter to use to import a graphic or equation. The conversion object must be of type `SR_OBJ_GRAPHIC` or `SR_OBJ_EQUATION`.

If you call this function before an object is converted, it supplies the import file hint string proposed for the object in `FrameMaker`.

If you call this function after an object is converted, it supplies the import file hint string that was assigned to the object in `FrameMaker`. The string returned by this function may be `NULL`. If the string is `NULL`, then `FrameMaker` determines the appropriate file format and filter to use when the graphic or equation is imported.

Returns

A string containing file format and filter information, or `NULL` if no format and filter are assigned for this object.

The syntax of a non-`NULL` string is:

```
<record_ver><vendor><format_id><platform><filter_ver><filter_name>
```

Each field in the string except `filter_name` specifies a 4-byte alphanumeric code. If a field code is less than four bytes, the extra bytes are padded with spaces. For example:

```
0001PGRFPICTMAC61.0 Built-in PICT Writer
```

For a complete description of possible string contents, see [“Sr_SetImportFileHint\(\)” on page 206](#).

If the string returned by this function is `NULL`, `FrameMaker` determines the appropriate file format and filter to use when the graphic or equation is imported.

Examples

The following code retrieves the import file hint for a conversion object:

```
. . .
StringT importHint;
. . .
importHint = Sr_GetImportFileHint(srObj);
. . .
```

See also

- [“Sr_SetImportFileHint\(\)” on page 206](#)
- [“SrConvObjT” on page 415](#)
- [“SRW_erno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_GetInsertedTablePartElementName()

Retrieves the element name for a specified table part automatically inserted into a FrameMaker document by a conversion object of type `SR_OBJ_TABLE` if a markup element for that table part is not available.

Synopsis

```
#include "fm_struct.h"
. . .
StringT Sr_GetInsertedTablePartElementName(SrConvObjT srObj,
      SrwTablePartTypeT tablePart);
```

Arguments

<i>srObj</i>	Conversion object to query
<i>tablePart</i>	Type of table part for which to return element name

Details

In FrameMaker, a table must have specific parts such as table body, table rows, and table cells. Also, each part must have a corresponding FrameMaker element; the EDD must be set up to provide these elements for the table parts. If the markup you import doesn't include corresponding table elements, FrameMaker automatically includes the required elements in the FrameMaker document.

This function returns the FrameMaker element name for the specified table part. If you call this function before the table is created in the FrameMaker document, it returns the element name of the table part proposed for insertion. If you call this function after the table is

created in the FrameMaker document, it returns the name of the table-part element inserted into the table.

To specify the name of a FrameMaker element for an table part, call `Sr_SetInsertedTablePartElementName()`.

Returns

A string containing the element name, or `NULL` on error. On error, `SRW_errno` is set to one of the following error codes:

- `SRW_E_INVALID_CONV_OBJ`
- `SRW_E_WRONG_OBJ_TYPE`

Examples

The following code retrieves the element name for the body of a table:

```
. . .
StringT tblPrtElemName;
. . .
tblPrtElemName = Sr_GetInsertedTablePartElementName (srObj,
    SRW_TABLE_BODY);
. . .
```

See also

- [“Sr_SetInsertedTablePartElementName\(\)” on page 209](#)
- [“SrConvObjT” on page 415](#)
- [“SrwTablePartTypeT” on page 403](#)

Sr_GetInsertLoc()

Returns the location in the FrameMaker document where the FrameMaker object associated with an import conversion object will be inserted.

Synopsis

```
#include "fm_struct.h"
. . .
SrInsertLocT Sr_GetInsertLoc(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

This function indicates where in the current FrameMaker document the specified conversion object's content will be inserted. This location corresponds to an insertion point in the document.

If you call this function before the object is converted, it returns the proposed insertion location. If you call this function after the object is converted, it reports where in the FrameMaker document the insertion occurred.

To specify the insertion location in a FrameMaker document for an import conversion object before it is converted, call `Sr_SetInsertLoc()`.

Returns

A structure that describes a document location, or a zeroed-out structure on error. On error, `SRW_errno` is set to one of the following possible codes:

- `SRW_E_BAD_OBJ_HANDLE`
- `SRW_E_INVALID_CONV_OBJ`

`SrInsertLocT` is defined as:

```
typedef struct {
    SrLocationT pos;
    union {
        F_ObjHandleT flowId;
        F_ObjHandleT mkrId;
        F_ElementLocT elemLoc;
    } u;
} SrInsertLocT;
```

For more information about `SrInsertLocT`, see the *FDK Programmer's Reference* and the *FDK Programmer's Guide*.

The possible values for `pos` are:

- `SR_LOC_FLOW`
`u` contains a flow ID that indicates the flow into which the first element should be inserted. Note that this is only valid when the proposed element is valid at the highest level in the flow.
- `SR_LOC_MARKER_TEXT`
`u` contains a marker ID that indicates the FrameMaker marker into which text should be inserted.
- `SR_LOC_ELEMENT`:
`u` contains an element location that indicates the location in the document where the new text, element, or object should be inserted. `F_ElementLocT` is defined as:

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

The parent element is the element that will contain the inserted element. The child element is the sibling of the inserted element that immediately follows the inserted element. If child is 0, the inserted element will be the last child of the parent.

The offset counts at what number of characters into the parent to insert the current element. (Note that an offset can also count into a child element, but only when dealing with a selection range. However, for this function, selection ranges never occur.)

- **SR_LOC_BOOK**

u is ignored, and the element is inserted into the current book.

Examples

The following code ensures the proposed insert location for a GLOSSARY conversion object is the first element position in it's parent element. For example, if ENDMATTER contains ENDNOTES and GLOSSARY, the code exports GLOSSARY as the first child in ENDMATTER.

```
. . .
SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    if (F_StrIEqual(eventp->u.tag.gi, (StringT) "GLOSSARY")) {
        SrInsertLocT insLoc;
        SrConvObjT docObj;
        F_ObjHandleT firstChildId, docId;

        docObj = Sr_GetCurConvObjOfType(SR_OBJ_DOC);
        if (!docObj)
            docObj = Sr_GetCurConvObjOfType(SR_OBJ_BOOK_COMP);
        if(docObj)
            docId = Sr_GetDocId(docObj);

        /* Make this element is the first child of its parent */
        insLoc = Sr_GetInsertLoc(srObj);

        insLoc.u.elemLoc.childId = F_ApiGetId(docId,
            insLoc.u.elemLoc.parentId, FP_FirstChildElement);
        insLoc.u.elemLoc.offset = 0;

        Sr_SetInsertLoc(srObj, &insLoc);
    }
    return(Sr_Convert(eventp, srObj));
}
. . .
```

See also

- [“Sr_SetInsertLoc\(\)” on page 210](#)
- [“SrInsertLocT” on page 416](#)
- [“SrConvObjT” on page 415](#)

Sr_GetLineBrkInfo()

Retrieves line-break information associated with the specified import conversion object of type `SR_OBJ_ELEM`.

Synopsis

```
#include "fm_struct.h"
. . .
IntT Sr_GetLineBrkInfo(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

This function indicates whether each record end (RE) for an element's content will be translated as a space or as a forced return.

The PCDATA that is contained by a markup element can be typed into the markup file as many lines of text, and each new-line character in the PCDATA is recognized as an RE. By default, FrameMaker translates any such RE as a space character. In other words, all the lines of text in the markup element get inserted in the FrameMaker element as one stream of text. The line length of this text is determined by FrameMaker's column width, and word-wrap properties.

Note that a forced return starts the text on a new line, but does not begin a new paragraph.

You should be aware that you can set up read/write rules to specify this line break info without writing an API client. This function provides a way to check the line break rule for the current element. You can change the line break setting via `Sr_SetLineBreakInfo()`.

If you call this function before the object is converted, it reports the proposed line-break setting for the object. If you call this function after the object is converted, it reports the line-break setting used by the object.

Returns

`SR_LB_SPACE`, `SR_LB_FORCED_RETURN`, or 0 on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`. (This may return `SRW_E_WRONG_OBJ_TYPE` on error.)

Examples

The following code retrieves the proposed line-break information for an event of type `SR_EVT_BEGIN_ELEM`, and if the corresponding element is tagged `CODE_SEGMENT`, it forces a new line for every record end in the element's content.

```
#include "fm_struct.h"

SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    SrConvObjT elemObj;
    SrEventT *elemEvt;
    IntT linebreak = 0;

    if (eventp->evtype == SR_EVT_BEGIN_ELEM) {
        if (F_StrIEqual(eventp->u.tag.gi,
                        (StringT) "CODE_SEGMENT")) {
            linebreak = Sr_GetLineBrkInfo(srObj);
            if (linebreak != SR_LB_FORCED_RETURN)
                Sr_SetLineBrkInfo(srObj, SR_LB_FORCED_RETURN);
        }
        return (Sr_Convert(eventp, srObj));
    }
}
```

See also

- [“Sr_SetLineBrkInfo\(\)” on page 213](#)
- [“SrConvObjT” on page 415](#)
- [“SRW_erno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `IntT`

Sr_GetObjType()

Retrieves the object type of the specified conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

SrObjTypeT Sr_GetObjType(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

This function is commonly used to test the a conversion object's type in an event handler in order to determine how to process the object before converting it. The conversion object types are defined by the `SrObjTypeT` data type. See [“SrObjTypeT” on page 395](#) for the list of object types.

Note that you cannot modify a conversion object's type.

Returns

The object type of the conversion object.

Examples

The following code tests the current conversion object to see if it is of type `SR_OBJ_BOOK`. If it is, code inside the `if` statement is executed.

```
. . .
SrConvObjT srObj;
. . .
if (Sr_GetObjType(srObj) == SR_OBJ_BOOK) {
    /*You know the specified conversion object indicates a book*/
    . . .
}
. . .
```

See also

- [“Sr_GetChildConvObj\(\)” on page 122](#)
- [“Sr_GetCurConvObj\(\)” on page 128](#)
- [“Sr_GetCurConvObjOfType\(\)” on page 129](#)
- [“Sr_GetParentConvObj\(\)” on page 153](#)
- [“SrConvObjT” on page 415](#)
- [“SrObjTypeT” on page 395](#)

Sr_GetParentConvObj()

Retrieves the import conversion object that is immediately superior to the specified conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

SrConvObjT Sr_GetParentConvObj(SrConvObjT childConvObj);
```

Arguments

childConvObj Conversion object to query

Details

As import events occur, corresponding conversion objects are created. These conversion object/event pairs are pushed onto a stack that represents the hierarchy of currently open conversion objects. Note that for container objects, import events come in pairs; for every BEGIN event to open a container there is a corresponding END event to close it. A conversion object is opened when the BEGIN event occurs, and is closed when the END event occurs.

Note that any number of nested conversion objects can be open at one time. However, among sibling objects only one sibling can be open at a time, because an object is removed from the stack as soon as it is closed.

```
<LEVEL_ONE>
  <LEVEL_TWO>
    <LEVEL_THREE>
      <LEVEL_FOUR>
```

Sr_GetParentConvObj() returns the ID of the the open conversion object that is immediately superior to the specified object. In the above example, if *LEVEL_THREE* is the specified object, *LEVEL_TWO* is its parent.

A use for this function might be to get the parent of a table. If the parent object is an element named *FINANCIAL_DATA*, you could apply a specific table format to the table.

In your event handler routines, you must declare a local variable of type *SrConvObjT* for the conversion object returned by this function.

Returns

The parent conversion object, or *NULL* if the specified conversion object is not subordinate to another conversion object.

Examples

The following code checks the current object type. If it is a table, it checks whether the parent object indicates a FINANCIAL_DATA element:

```
. . .
SrConvObjT srObj, containerObj;
SrEventT *containerp;

. . .
if (Sr_GetObjType(srObj) == SR_OBJ_TABLE) {
    containerObj = Sr_GetParentConversionObj(srObj);
    containerp = Sr_GetAssociatedEvent(containerObj);
    if (F_StrIEqual(containerp->u.tag, (StringT)"FINANCIAL_DATA"){
        /*Set the table format accordingly*/
    }
. . .
```

See also

- [“Sr_GetChildConvObj\(\)” on page 122](#)
- [“Sr_GetCurConvObj\(\)” on page 128](#)
- [“SrConvObjT” on page 415](#)

Sr_GetPrevStructuredGi()

Deprecated: Sr_GetPrevSgmlGi()

Returns the *generic identifier (GI)* for the previous sibling of the specified conversion object.

Synopsis

```
#include "fm_struct.h"

. . .
StringT Sr_GetPrevStructuredGi(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

To handle content-coded tables, an import conversion object for an element stores the element *GI* corresponding to the most recent sibling conversion object popped off the conversion object stack. Call `Sr_GetPrevStructuredGi()` to get the name of that element.

Returns

The *GI* of the previous sibling, or `NULL` if there is no previous sibling.

Examples

The following code retrieves the *GI* for the previous sibling of the current conversion object:

```
. . .
StringT PrevGi;
. . .
PrevGi = Sr_GetPrevStructuredGi(srObj);
. . .
F_ApiDeallocateString(&PrevGi);
. . .
```

See also

- [“Sr_GetChildConvObj\(\)” on page 122](#)
- [“Sr_GetParentConvObj\(\)” on page 153](#)
- [“SrConvObjT” on page 415](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_GetPrivateData()

Returns a pointer to data your application has associated with the specified conversion object. This pointer is the address of an arbitrary data type that you have allocated, initialized, and maintained.

Synopsis

```
#include "fm_struct.h"
. . .
PtrT Sr_GetPrivateData(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

You can allocate and maintain data that is private to your application, and then assign the address of that data to a specific conversion object. This is a way to save a state within your application, and then associate it with a specific level in the hierarchy of the conversion object stack. This function retrieves a pointer to such data that was previously set to the specified conversion object.

Note that your application owns the data. Also, the data can be of any addressable type, from a single variable to a structure within a deep hierarchy of structures. It is up to your application to maintain the data content.

Important: When the conversion object is closed, it is removed from the conversion object stack and its allocated memory is freed. However, the data referenced by the private data pointer is not deallocated. If the only reference to this data is stored with the conversion object, and that object is closed, you will not be able to deallocate the private data. Be sure to trap ending events (for example, `SR_EVT_END_ELEM`) and check their associated objects for private data pointers.

Returns

A pointer to private data, or `NULL` if no private data is set for the object.

Examples

The following code declares a structure that will be used to keep data about the element `TABLE`. It initializes the structure, and passes the address of the structure to `Sr_SetPrivateData()`, which assigns the structure to the `TABLE` element.

If the current element is `ROW`, the code loops through the element's parents until it finds a `TABLE` element. It then retrieves the private data for the `TABLE` element. Notice that the pointer for the data structure must be declared globally to the event handler. Then you can trap the `END ELEMENT` event for the table to free the structure's memory.

```
. . .
typedef struct {
    IntT rowCount;
    BoolT isFinancial;
} MyDataT;

SrErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    SrEventT *parEvt;
    SrConvObjT parentObj;
    MyDataT *testData;
```

```
if (eventp->evtype == SR_EVT_BEGIN_ELEM) {
    if (F_StrIEqual(eventp->u.tag.gi, "TABLE")) {
        testData = F_Alloc(sizeof(MyDataT), DSE);
        testData->rowCount = (IntT)4;
        testData->isFinancial = (BoolT)True;
        Sr_SetPrivateData(srObj, testData);
    } else if (F_StrIEqual(eventp->u.tag.gi, "ROW")) {
        parentIbj = srObj;
        do {
            parentObj = Sr_GetParentConvObj(srObj);
            parEvt = Sr_GetAssociatedEvent(parentObj);
        } while(parEvt && !F_StrIEqual(
            parEvt->u.tag.gi, (ConstringT)"TABLE"))
        if(parEvt && (testData = Sr_GetPrivateData(parentObj)))
        {
            F_Printf(NULL, (StringT)
                "Private Data for TABLE is %d and %d",
                testData->rowCount, testData->isFinancial);
        }
    }
}
if (eventp->evtype == SR_EVT_END_ELEM) {
    if (F_StrIEqual(eventp->u.tag.gi, "TABLE")) {
        testData = Sr_GetPrivateData(srObj);
        F_Free(testData);
    }
}
return(Sr_Convert(eventp, srObj));
}
```

See also

- [“Sr_SetPrivateData\(\)” on page 215](#)
- [“SrConvObjT” on page 415](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `PtrT`

Sr_GetProcessingFlags()

Returns the status of the element processing flags that indicate the processing for an import conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

IntT Sr_GetProcessingFlags(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

If you call this function before an object is converted, it reports the proposed processing for the object. You can modify this proposal by calling `Sr_SetProcessingFlags()`. Note that if the conversion object is not for an element, this function returns 0.

If you call this function after an object is converted, it reports the processing that was performed on the object.

Returns

A status integer set to one of the following values:

- 0
This indicates either the object will be converted normally, or it is not an element.
- `SRW_UNWRAP_ELEMENT`
This indicates the `unwrap` read/write rule was specified for the given element.
- `SRW_DROP_ELEMENT_CONTENT`
This indicates the `drop content` read/write rule was specified for the given element.
- `SRW_UNWRAP_ELEMENT | SRW_DROP_ELEMENT_CONTENT`
This indicates both the `unwrap` and `drop content` read/write rules were specified for the given element.

Examples

The following code makes sure to remove any secret element and drop it's content.

```
#include "fm_struct.h"

SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
```

```
{
F_AttributeT importAttribute;
IntT flags;
SrwErrorT errorStatus;

if (eventp->evtype == SR_EVT_BEGIN_ELEM) {
    if (F_StrIEqual(eventp->u.tag.gi, "MY_ELEMENT")) {
        importAttribute = Sr_GetAttrVal(srObj, (StringT) "access");
        /*Now check whether the attribute value is SECRET*/
        if (F_StrIEqual(importAttribute.values.val[0],
                        (StringT) "SECRET")) {
            flags = Sr_GetProcessingFlags(srObj);
            if (!(flags & SRW_DROP_ELEMENT_CONTENT) &&
                (flags & SRW_UNWRAP_ELEMENT))) {
                flags = (SRW_DROP_ELEMENT_CONTENT | SRW_UNWRAP_ELEMENT);
                errorStatus = Sr_SetProcessingFlags(srObj, flags);
            }
        }
        F_ApiDeallocateAttribute(&importAttribute);
    }
}
return (Sr_Convert(eventp, srObj));
}
```

See also

- [“Sr_SetProcessingFlags\(\)” on page 216](#)
 - [“SrConvObjT” on page 415](#)
 - [“Primitive data types” on page 383](#) for more information on StringT for more information
- IntT

Sr_GetPropVal()

Retrieves a specified FrameMaker property for an import conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
StringT Sr_GetPropVal(SrConvObjT srObj,
SrwFmPropertyT fmProp);
```

Arguments

<i>srObj</i>	Conversion object to query
<i>fmProp</i>	Property to examine

Details

Conversion objects can have properties associated with them. The properties that can be assigned to an object depend on the conversion object type. The objects that can have properties are of the following types:

SR_OBJ_ELEM	SR_OBJ_TABLE	SR_OBJ_TABLE_TITLE
SR_OBJ_TABLE_HEADING	SR_OBJ_TABLE_BODY	SR_OBJ_TABLE_FOOTING
SR_OBJ_TABLE_ROW	SR_OBJ_TABLE_CELL	SR_OBJ_TABLE_COLSPEC
SR_OBJ_TABLE_SPANSPEC	SR_OBJ_GRAPHIC	SR_OBJ_EQUATION
SR_OBJ_VARIABLE	SR_OBJ_MARKER	SR_OBJ_XREF
SR_OBJ_FOOTNOTE		

For a list of the properties that apply the these object types, see [“SrwFmPropertyT” on page 397](#).

Note that the property value is a string, in spite of the fact that some of the properties are enumerated types or other numeric values. These numeric values are handled as strings to unify the format of all conversion object properties.

Important: When the string returned by this function is no longer needed, call `F_ApiDeallocateString()` to free it.

If you call this function before an object is converted, it reports the proposed value for the property. You can change this value with `Sr_SetPropVal()`. If you call it after an object is converted, it reports the value used for conversion. Once converted, this value cannot be changed.

To retrieve a list of all property value data structures for an import conversion object, call `Sr_GetPropVals()`.

Returns

A string containing the value of the specified property, or `NULL` on error. On error, `SRW_errno` is set to one of the following possible values:

- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_BAD_VALUE`
- `SRW_E_OBJ_HAS_NO_SUCH_PROP`

Examples

The following code traps an xref event and gets the xref format property value. The code then sets a different cross-reference format.

```
#include "fm_struct.h"
#include "fstring.h"

SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    StringT xrefFmt;

    if(eventp->evtype == SR_EVT_BEGIN_ELEM) {
        if(Sr_GetObjType(srObj) == SR_OBJ_XREF) {
            xrefFmt = Sr_GetPropVal(srObj, SRW_PROP_XREF_FORMAT);
            if (F_StrIsEmpty(xrefFmt))
                /* FrameMaker doesn't have a r/w rule or */
                /* any indication in the EDD to set an xref format. */
                Sr_SetPropVal(srObj, SRW_PROP_XREF_FORMAT,
                              (StringT) "Page");
        }
        else
            /* This command will override whatever xref format */
            /* was indicated by the current struct app. */
            Sr_SetPropVal(srObj, SRW_PROP_XREF_FORMAT,
                          (StringT) "Heading & Page");
        F_ApiDeallocateString(&xrefFmt);
    }
}

Sr_Convert(eventp, srObj);
}
```

See also

- [“Sr_SetPropVal\(\)” on page 217](#)
- [“SrConvObjT” on page 415](#)
- [“SrwFmPropertyT” on page 397](#)
- [“SRW_erno” on page 418](#)

Sr_GetPropVals()

Retrieves a list of properties for the specified conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
SrwPropValsT Sr_GetPropVals(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

Conversion objects can have properties associated with them. The properties that can be assigned to an object depend on the conversion object type. The objects that can have properties are of the following types:

SR_OBJ_ELEM	SR_OBJ_TABLE	SR_OBJ_TABLE_TITLE
SR_OBJ_TABLE_HEADING	SR_OBJ_TABLE_BODY	SR_OBJ_TABLE_FOOTING
SR_OBJ_TABLE_ROW	SR_OBJ_TABLE_CELL	SR_OBJ_TABLE_COLSPEC
SR_OBJ_TABLE_SPANSPEC	SR_OBJ_GRAPHIC	SR_OBJ_EQUATION
SR_OBJ_VARIABLE	SR_OBJ_MARKER	SR_OBJ_XREF
SR_OBJ_FOOTNOTE		

For a list of the properties that apply the these object types, see [“SrwFmPropertyT” on page 397](#).

This function returns `SrwPropValsT`, which is a list of all the properties for a specified conversion object. `SrwPropValsT` is defined as:

```
typedef struct {
    IntT len; /* Number of properties in array. */
    SrwPropValT *vals; /* Pointer to array of values. */
} SrwPropValsT;
```

Each property is expressed by a `SrwPropValT` structure, which is defined as:

```
typedef struct {
    SrwFmPropertyT prop;
    StringT value;
} SrwPropValT;
```

Note that the property value is a string, in spite of the fact that some of the properties are enumerated types or other numeric values. These numeric values are handled as strings to unify the format of all conversion object properties.

Important: Note that actual property values are strings. If you want to store a value in a variable, then you must either use `Sr_GetPropVal()`, or else use `F_StrCopyString()`, as follows:

```
propVal = F_StrCopyString(propVals.vals[i].value).
```

Important: When you are finished with the list of property values, be sure to call `Srw_DeallocatePropVals()` to free its memory. Use `Srw_DeallocatePropVal()` to free up a single property value.

If you call this function before an object is converted, it reports the proposed list of property values to use for conversion. If you call this function after the object was converted, it returns the list of properties that was used. To specify a different list of property values for an object before it is converted, call `Sr_SetPropVals()`. You cannot change the property values after the object is converted.

To retrieve the string that represents a single property value, call `Sr_GetPropVal()`.

Returns

The property values list associated with the conversion object, or a zeroed-out data structure on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code traps an xref object, gets its list of properties, and loops through the list. It then sets the `SrwPropValsT` structure, in case it changed the xref format.

```
#include "fm_struct.h"
#include "fstrings.h"

SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    SrwPropValsT props;
    UIntT i;

    if (eventp->evtype == SR_EVT_BEGIN_ELEM) {
        if (Sr_GetObjType(srObj) == SR_OBJ_XREF) {
            props = Sr_GetPropVals(srObj);
            for (i = 0; i < props.len; i++) {
```

```
        switch (props.val[i].prop) {
            case SRW_PROP_XREF_ID:
                F_Printf(NULL, (StringT) "The xref ID is %s\n",
                        props.val[i].value);
                break;
            case SRW_PROP_XREF_FORMAT:
                F_Printf(NULL,
                        (StringT) "The xref format is %s\n",
                        props.val[i].value);
                if(F_StrIEqual(props.val[i].value,
                        (StringT) "Heading & Page")) {
                    F_StrCpy(props.val[i].value, (StringT)"Page");
                }
                break;
            default:
                break;
        } /* end of switch */
    } /* end of for loop */
    Sr_SetPropVals(srObj, &props);
    Srw_DeallocatePropVals(&props);
} /* end of IF XREF */
} /* end of BEGIN_ELEM */
Sr_Convert(eventp, srObj);
}
```

See also

- [“Sr_SetPropVals\(\)” on page 219](#)
- [“Sr_GetPropVal\(\)” on page 159](#)
- [“SrConvObjT” on page 415](#)
- [“SrwPropValsT” on page 421](#)
- [“SRW_erno” on page 418](#)

Sr_GetRefElemTag()

Retrieves the name of the FrameMaker reference element associated with the specified conversion object of type `SR_OBJ_REF_ELEM`. Objects of this type are proposed when importing an SDATA entity.

Synopsis

```
#include "fm_struct.h"
. . .
StringT Sr_GetRefElemTag(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

In markup documents, SDATA entities are used to represent information that might be referenced a number of times in the markup document. These can require processing, such as a current date, or they can be static data such as copyright boilerplate or a graphic for the company logo. FrameMaker can handle such entities in various ways. The current date is best imported as a FrameMaker date variable. Likewise, simple boilerplate could be imported as a user-defined variable, or complex boilerplate might be imported as a text inset. Variables must be set up in the import template, and your structure application must include read/write rules to convert instances of the specific entity as the correct variable.

You can also import an SDATA entity as almost any set of document objects. For example, an entity can be imported as one or more anchored frames. To do this, you map an SDATA entity to an element that you store on a reference page of your import template. You then use read/write rules to map the entity to the reference element. For more information, see “Translating SDATA entities as FrameMaker reference elements” in the *FrameMaker Structure Application Developer's Guide*.

It is possible that you might want two or more reference elements for a given SDATA entity. For example, you might have a large company logo for instances within regular body text, and a smaller logo for instances within tables, each wrapped in differently named reference elements. You can only specify one of these logo elements via read/write rules. With `Sr_GetRefElemTag()` you can check to see what the proposed reference element is, and with `Sr_SetRefElemTag()` you can change that proposal.

If you call this function before an object is converted, it reports the name of the proposed reference element to use on conversion. If you call this function after the object is converted, it reports the name of the reference element that was used.

Returns

The name of a reference element, or `NULL` on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code traps an entity event, and checks for a proposed reference element named `BIG_LOGO`. If `BIG_LOGO` is proposed, then it checks to see if a table object is currently open. If a table is open, the code substitutes `SMALL_LOGO`.

```
#include "fm_struct.h"

SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    StringT refElemName;

    if (eventp->evtype == SR_EVT_BEGIN_ENTITY) {
        refElemName = Sr_GetRefElemTag(srObj);
        if (F_StrIEqual(refElemName, (StringT) "BIG_LOGO")) {
            if (Sr_GetCurConvObjOfType(SR_OBJ_TABLE))
                Sr_SetRefElemTag(srObj, (StringT) "SMALL_LOGO");
        }
    }
    F_ApiDeallocateString(&refElemName);
    Sr_Convert(eventp, srObj);
}
```

See also

- [“Sr_SetRefElemTag\(\)” on page 221](#)
- [“Sr_GetFmElemTag\(\)” on page 140](#)
- [“SrConvObjT” on page 415](#)
- [“SRW_erno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_GetSessionProps()

Retrieves import session properties for the current session (as specified by a conversion object of type `SR_OBJ_SESSION`).

Synopsis

```
#include "fm_struct.h"

. . .

SrSessionPropsT Sr_GetSessionProps(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

To retrieve the current session properties, you should first call `Sr_GetCurConvObjOfType(SR_OBJ_SESSION)` to get the Id of the current session object, then pass that Id to `Sr_GetSessionProps()`.

To set or change import session properties, call `Sr_SetSessionProps()`.

Important: Be sure to deallocate the `SrSessionPropsT` returned by this call when the structure is no longer needed.

Returns

An `SrSessionPropsT` structure describing current import session property settings. Note that the table ruling style is initially specified via the read/write rule, table ruling style is.

`SrSessionPropsT` is defined as:

```
typedef struct {
    BoolT isBatchMode; /*True if the current session is a batch.
                       NOTE: This field is Read Only.*/
    StringT tableRulingStyle; /*Name of a table ruling style*/
    BoolT overwriteFiles; /*True if batch can overwrite files of
                          same name in the target directory*/
} SrSessionPropsT;
```

Examples

The following case statement prints out the import session properties. Finally, the code frees the space allocated to the `SrSessionPropsT` structure:

```
. . .
SrSessionPropsT importSessionProps;
SrConvObjT sessionObj;
. . .
```

```
case SR_EVT_BEGIN_READER:
    Sr_Convert(eventp, srObj);
    sessionObj = Sr_GetCurConvObjOfType(SR_OBJ_SESSION);
    importSessionProps = Sr_GetSessionProps(sessionObj);
    if(importSessionProps.isBatchMode)
        F_Printf(NULL, "\nImporting in batch mode");
    else
        F_Printf(NULL, "\nNOT importing in batch mode");
    if(importSessionProps.overwriteFiles)
        F_Printf(NULL, "\nOverwriting files");
    else
        F_Printf(NULL, "\nNOT overwriting files");
    F_Printf(NULL, "\nTable ruling style: %s",
            importSessionProps.tableRulingStyle);
    Sr_DeallocateSessionProps(&importSessionProps);
    break;
. . .
```

See also

- [“Sr_SetSessionProps\(\)” on page 222](#)
- [“Sr_DeallocateSessionProps\(\)” on page 105](#)
- [“Sr_GetCurConvObjOfType\(\)” on page 129](#)
- [“SrConvObjT” on page 415](#)
- [“SrSessionPropsT” on page 417](#)

Sr_GetSpanSpecs()

Retrieves a list of *CALS* span specifications (*spanspecs*) for a table, table heading, table body, or table footing import conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

SrwSpanSpecsT Sr_GetSpanSpecs(SrConvObjT srObj);
```

Arguments

convObj Conversion object to query

Details

To retrieve the list of *spanspecs* for a table or table part associated with an import conversion object, call *Sr_GetSpanSpecs()*. Use this function only with *CALS* tables. *srObj* must be one of the following types:

- *SR_OBJ_TABLE*
- *SR_OBJ_TABLE_HEADING*
- *SR_OBJ_TABLE_BODY*
- *SR_OBJ_TABLE_FOOTING*

Note that in a strict *CALS* model, calling *Sr_GetSpanSpecs()* with an object of type *SR_OBJ_TABLE_BODY* always returns an empty data structure.

Important: Ordinarily, *Sr_GetSpanSpecs()* is not called for the current conversion object, but for its parent object. For example, if the current object is of type *SR_OBJ_TABLE*, then the object's *spanspecs* have not yet been set. On the other hand, when the current object is of type *SR_OBJ_TABLE_ROW* or *SR_OBJ_TABLE_CELL*, there might be good reason to call *Sr_GetSpanSpecs()* on the parent table object to determine column information for the row or cell.

If you call *Sr_GetSpanSpecs()* before converting an object, it returns the proposed list of *spanspecs* to insert into the FrameMaker document. If you call this function after converting an object, it returns the list of *spanspecs* that was used to create the corresponding FrameMaker table or table part.

After it is no longer needed use *Srw_DeallocateSpanSpecs()* to deallocate the *SrwSpanSpecsT* structure returned by this function.

Returns

An *SrwSpanSpecsT* structure, or a zeroed-out data structure on error. On error, *SRW_errno* is set to *SRW_E_WRONG_OBJ_TYPE*.

Examples

The following code demonstrates a call to `Sr_GetSpanSpecs()`:

```
. . .
SrwErrorT DetermineTableSpanSpecs(SrConvObjT srObj)
{
    SrwSpanSpecsT spanspecs;
    SrConvObjT tblObj;
    switch(Sr_GetObjType(srObj))
    {
        case SR_OBJ_TABLE_ROW:
            /* Retrieve the current table object */
            tblObj = Sr_GetCurConvObjOfType(SR_OBJ_TABLE);
            /* Retrieve copy of spanspecs for table object. */
            spanspecs = Sr_GetSpanSpecs(tblObj);
            . . .
            Srw_DeallocateSpanSpecs(&spanspecs);
    }
    . . .
}
```

See also

- [“Sr_SetSpanSpecs\(\)” on page 224](#)
- [“Srw_DeallocateSpanSpecs\(\)” on page 255](#)
- [“SrConvObjT” on page 415](#)
- [“SrwSpanSpecsT” on page 423](#)
- [“SRW_errno” on page 418](#)

Sr_GetStraddles()

Retrieves a list of running straddles for a table import conversion object created by a `start vertical straddle read/write` rule.

Synopsis

```
#include "fm_struct.h"
. . .
SrwStraddlesT Sr_GetStraddles(SrConvObjT srObj);
```

Arguments

convObj Conversion object to query

Details

To retrieve the list of running straddles for a table associated with an import conversion object, call `Sr_GetStraddles()`. *srObj* must be of type `SR_OBJ_TABLE`.

Important: Ordinarily, `Sr_GetStraddles()` is not called for the current conversion object, but for its parent object. For example, if the current object is of type `SR_OBJ_TABLE`, then the object's running straddles have not yet been set. On the other hand, when the current object is of type `SR_OBJ_TABLE_ROW` or `SR_OBJ_TABLE_CELL`, there might be good reason to call `Sr_GetStraddles()` on the parent table object to determine column information for the row or cell.

You can set or change the list of straddles associated with a conversion object by calling `Sr_SetStraddles()`.

After it is no longer needed, call `Srw_DeallocateStraddles()` to deallocate the `SrwStraddlesT` structure returned by this function.

Returns

An `SrwStraddlesT` structure, or a zeroed-out data structure on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

`SrwStraddlesT` is defined as:

```
typedef struct {
    IntT len; /* Number of straddles in array. */
    SrwStraddleT *val; /* Pointer to array of straddles. */
} SrwStraddlesT;
```

`SrwStraddleT` is defined as:

```
typedef struct {
    F_ObjHandleT cellId;
    StringT straddleName;
} SrwStraddleT;
```

Examples

The following code demonstrates a call to `Sr_GetStraddles()`:

```
. . .
SrwErrorT DetermineTableStraddles(SrConvObjT srObj)
{
    SrwStraddlesT straddles;
    SrConvObjT tblObj;
    switch(Sr_GetObjType(srObj))
    {
        case SR_OBJ_TABLE_ROW:
            /*Retrieve the current table object. */
            tblObj = Sr_GetCurConvObjOfType(SR_OBJ_TABLE);
            /* Retrieve copy of straddles for table object. */
            straddles = Sr_GetStraddles(tblObj);
            . . .
            Srw_DeallocateStraddles(&straddles);
    }
    . . .
}
```

See also

- [“Sr_SetStraddles\(\)” on page 225](#)
- [“Srw_DeallocateStraddles\(\)” on page 257](#)
- [“SrConvObjT” on page 415](#)
- [“SrwStraddlesT” on page 424](#)
- [“SRW_errno” on page 418](#)

Sr_GetTableCellStartsNewRow()

Determines whether a table cell element of type `SR_OBJ_TABLE_CELL` should use an available empty cell in the current row, or force a new row to be created.

Synopsis

```
#include "fm_struct.h"
. . .
BoolT Sr_GetTableCellStartsNewRow(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

Call `Sr_GetTableCellStartsNewRow()` to determine whether a table cell element of type `SR_OBJ_TABLE_CELL` should use an available empty cell in the current row of a table or force a new row to be created for the cell.

To specify whether a table cell element should use an available cell or force creation of a new row, call `Sr_SetTableCellStartsNewRow()`.

Returns

1 if a table cell element should force a new row; 0, otherwise. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code determines if a table cell element should start a new row in a table:

```
. . .
BoolT UseNewRow;
. . .
UseNewRow = Sr_GetTableCellStartsNewRow(srObj);
. . .
```

See also

- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `BoolT`

Sr_GetTemplateDocId()

Retrieves the document ID for the FrameMaker import template used to generate FrameMaker document objects. The import template is specified as part of the current structure application. All available structure applications for the current FrameMaker session are specified in the application definition file. For more information about the application file, see “Chapter 4, Working with Special Files” in the *FrameMaker Developer’s Guide*.

Synopsis

```
#include "fm_struct.h"
. . .
F_ObjHandleT Sr_GetTemplateDocId(VoidT);
```

Arguments

None.

Details

FrameMaker only opens the template document for import operations, so there never is a handle for the template document during export events. You can get the proposed template document ID before converting the `SR_EVT_BEGIN_READER` event. After you get the ID, you can use standard FDK commands to modify the document. For example, to completely change the template, you can open another document and import its formats and element catalog into the template document.

Returns

The ID of the FrameMaker template document.

Examples

The following code returns the document ID for the FrameMaker template used for importing and formatting objects inserted into a FrameMaker document:

```
. . .
F_ObjHandleT currentTemplateId;
. . .
if(eventp->evtype == SR_EVT_BEGIN_READER) {
    currentTemplateId = Sr_GetTemplateDocId();
    if (currentTemplateId) {
        /*Perform some action on the template document*/
    }
}
. . .
```

See also

- [“Sr_GetBookId\(\)” on page 119](#)
- [“Sr_GetDocId\(\)” on page 132](#)
- [“Srw_GetImportTemplateFilePath\(\)” on page 266](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `F_ObjHandleT`

Sr_GetTextInsetFilePath()

Retrieves the platform-independent filepath for a FrameMaker text inset associated with an import conversion object of type `SR_OBJ_TEXT_INSET`.

Synopsis

```
#include "fm_struct.h"
. . .
FilePathT *Sr_GetTextInsetFilePath(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

To determine the platform-independent filepath used for a FrameMaker text inset associated with an `SR_OBJ_TEXT_INSET` conversion object, call `Sr_GetTextInsetFilePath()`. This is the filepath used either to import a copy of the text inset into the FrameMaker

document (insert by copy), or as the inset reference inserted into the document (insert by reference).

If you call `Sr_GetTextInsetFilePath()` before converting an object, it returns the proposed filepath for the text inset. To specify a different filepath for the inset, call `Sr_SetTextInsetFilePath()` before converting the object.

If you call `Sr_GetTextInsetFilePath()` after converting an object, the function returns the filepath that was used to create the text inset in the FrameMaker document.

`FilePathT` is a pointer to a platform-independent representation of a platform-specific path. For more information about working with platform-independent representations of filepaths, see the *FDK Programmer's Reference*. Use `F_FilePathFree()` to free the filepath when you have done with it.

Note that for external SDATA entities you can usually get the file path via `Srw_EntityHandler()`. However, if the entity refers to an external FrameMaker document, you must use the import handler `Sr_EventHandler()` to get the file path.

Returns

A pointer to a platform-independent filepath, or `NULL` on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code prints out the text inset file paths for any entity events that have associated text insets. Because not all entity events require text insets, you must test for the result of `Sr_GetTextInsetFilePath()`.

```
. . .
FilePathT *textInsetPath;
StringT s;
. . .
    case SR_EVT_BEGIN_ENTITY:
        if(textInsetPath = Sr_GetTextInsetFilePath(srObj)) {
            s = F_FilePathToPathName(textInsetPath, FDefaultPath);
            F_Printf(NULL, "\nText inset path: %s", s);
            F_ApiDeallocateString(&s);
            F_FilePathFree(textInsetPath);
        }
        break;
. . .
```

See also

- [“Sr_SetTextInsetFilePath\(\)” on page 228](#)
- [“Sr_GetTextInsetFlowTag\(\)” on page 176](#)
- [“Sr_GetTextInsetPageSpace\(\)” on page 179](#)

- [“Srw_EntityHandler\(\)” on page 259](#)
- [“SrConvObjT” on page 415](#)
- [“FilePathT” on page 409](#)

Sr_GetTextInsetFlowTag()

Retrieves the name of the FrameMaker text flow for a text inset associated with an import conversion object of type `SR_OBJ_TEXT_INSET`.

Synopsis

```
#include "fm_struct.h"
. . .
StringT Sr_GetTextInsetFlowTag(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

To determine the name of a FrameMaker text flow for a text inset associated with an `SR_OBJ_TEXT_INSET` conversion object, call `Sr_GetTextInsetFlowTag()`.

If you call this function before converting an object, it returns the proposed FrameMaker flow for the text inset. To specify a different flow tag for the text inset, call `Sr_SetTextInsetFlowTag()`.

Note that you can usually get information about external SDATA entities via `Srw_EntityHandler()`. However, if the entity refers to an external FrameMaker document, you must use the import handler `Sr_EventHandler()`.

Returns

The name of a flow, or `NULL` on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code checks for a specific entity you have set up to be treated as a text inset referring to a FrameMaker document. When it finds that entity, the code prints out the inset file path and the flow tag.

```
. . .
FilePathT *textInsetPath;
StringT s1, s2;
. . .
```



```
case SR_EVT_BEGIN_ENTITY:
    if(F_StrIEqual((StringT)"MyEnt", eventp->u.entname)) {
        textInsetPath = Sr_GetTextInsetFilePath(srObj);
        s1 = F_FilePathToPathName(textInsetPath, FDefaultPath);
        s2 = Sr_GetTextInsetFlowTag(srObj);
        F_Printf(NULL, "\nFm file path: %s and flow %s",
                  s1, s2);
        F_ApiDeallocateString(&s1);
        F_ApiDeallocateString(&s2);
        F_FilePathFree(textInsetPath);
    }
    break;
. . .
```

See also

- [“Sr_SetTextInsetFlowTag\(\)” on page 231](#)
- [“Sr_GetTextInsetFilePath\(\)” on page 174](#)
- [“Sr_GetTextInsetPageSpace\(\)” on page 179](#)
- [“SrConvObjT” on page 415](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_GetTextInsetFormatting()

Indicates how the content is formatted in a text inset associated with an import conversion object of type `SR_OBJ_TEXT_INSET`.

Synopsis

```
#include "fm_struct.h"
. . .
IntT Sr_GetTextInsetFormatting(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

Call `Sr_GetTextInsetFormatting()` to determine whether the content of a text inset associated with an `SR_OBJ_TEXT_INSET` conversion object is formatted according to the source document, according to the importing document, or formatted as plain text.

If you call this function before converting an object, it returns the proposed formatting for the text inset. To change the formatting of a proposed text inset, call

`Sr_SetTextInsetFormatting()`.

If you call `Sr_GetTextInsetFormatting()` after converting an object, it returns the formatting used for the text inset in the FrameMaker document.

Returns

`FV_SourceDoc`, `FV_EnclosingDoc`, `FV_PlainText`, or 0 on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code retrieves the formatting proposed for the text inset associated with the current conversion object:

```
. . .
FilePathT *textInsetPath;
IntT fmt;
. . .
case SR_EVT_BEGIN_ENTITY:
    if(textInsetPath = Sr_GetTextInsetFilePath(srObj)) {
        fmt = Sr_GetTextInsetFormatting(srObj);
        if(fmt == FV_SourceDoc)
            F_Printf(NULL, "\nInset formatted via source doc");
        else if(fmt == FV_EnclosingDoc)
            F_Printf(NULL, "\nInset formatted via enclosing
doc");
        else if(fmt == FV_PlainText)
            F_Printf(NULL, "\nInset formatted as plain text");
        F_FilePathFree(textInsetPath);
    }
    break;
. . .
```

See also

- [“Sr_SetTextInsetFormatting\(\)” on page 232](#)
- [“Sr_GetTextInsetFilePath\(\)” on page 174](#)
- [“Sr_GetTextInsetFlowTag\(\)” on page 176](#)
- [“SrConvObjT” on page 415](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `IntT`

Sr_GetTextInsetPageSpace()

Indicates whether a text inset associated with an import conversion object of type `SR_OBJ_TEXT_INSET` is associated with a FrameMaker body page or reference page.

Synopsis

```
#include "fm_struct.h"

. . .

IntT Sr_GetTextInsetPageSpace(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

Call `Sr_GetTextInsetPageSpace()` to determine whether a text inset associated with an `SR_OBJ_TEXT_INSET` conversion object is associated with a body page or reference page in a FrameMaker document.

If you call this function before converting an object, it returns the proposed page space for the text inset. To change the page association of a proposed text inset, call `Sr_SetTextInsetPageSpace()`.

If you call `Sr_GetTextInsetPageSpace()` after converting an object, it returns the page space used to create the text inset in the FrameMaker document.

Returns

`FV_BodyPage`, `FV_ReferencePage`, or 0 on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`. `FV_BodyPage` and `FV_ReferencePage` are defined in `fapidef.h`.

Examples

The following code retrieves the page space for the text inset associated with the current conversion object:

```
. . .
FilePathT *textInsetPath;
IntT pageSpace;
. . .
```

```
case SR_EVT_BEGIN_ENTITY:
    if(textInsetPath = Sr_GetTextInsetFilePath(srObj)) {
        pageSpace = Sr_GetTextInsetPageSpace(srObj);
        if(pageSpace == FV_BodyPage)
            F_Printf(NULL, "\nInset to a FrameMaker body page");
        else if(pageSpace == FV_ReferencePage)
            F_Printf(NULL, "\nInset to a FrameMaker ref page");
        else
            F_Printf(NULL, "\nInset NOT to a FrameMaker doc");
        F_FilePathFree(textInsetPath);
    }
    break;
. . .
```

See also

- [“Sr_SetTextInsetPageSpace\(\)” on page 234](#)
- [“Sr_GetTextInsetFilePath\(\)” on page 174](#)
- [“Sr_GetTextInsetFlowTag\(\)” on page 176](#)
- [“SrConvObjT” on page 415](#)
- [“SRW_erno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `IntT`

Sr_GetVariableName()

Retrieves the name of the FrameMaker nonelement variable associated with the specified import conversion object of type `SR_OBJ_VARIABLE`.

Synopsis

```
#include "fm_struct.h"
. . .
StringT Sr_GetVariableName(SrConvObjT srObj);
```

Arguments

srObj Conversion object to query

Details

This function retrieves the name of a FrameMaker nonelement variable associated with an import conversion object of type `SR_OBJ_VARIABLE`. The list of possible variable names is determined by the variables that are defined in the import template. For information about user defined variables and system variables in documents, see the *FrameMaker User's Guide*.

You can use read/write rules to specify that certain entities will be imported as FrameMaker variables. Usually, these rules are sufficient to convert the entity to a variable of a given name. In cases where the read/write rules are insufficient, you can use `Sr_GetVariableName()` and `Sr_SetVariableName()` to check and to change the variable name for an import object.

For example, you might have an entity for DEPT_NAME. Assume FrameMaker creates a variable called DeptName, and defines it as the fully expressed department name. Whenever this entity appears in a table, you might want a department abbreviation. In that case, you can create a variable in the import template called DeptNameAbbr, and define it as the abbreviated department name. Then you can trap the DEPT_NAME entity, check to see if a table is currently open, and if so, use the DeptNameAbbr variable.

If you call this function before converting an object, it returns a proposed FrameMaker variable name. If you call it after converting the object, it returns the variable name that was used.

To specify a different variable name, call `Sr_SetVariableName()`.

Returns

A variable name, or NULL on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code checks the name of the proposed variable, and if the variable is to be inserted in a copyright section, changes the variable name:

```
#include "fm_struct.h"

SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    SrEventT *parentEvent;
    SrConvObjT parentElem;
    StringT varName;
```

Sr_RowInUse()

```
if (eventp->evtype == SR_EVT_BEGIN_ENTITY) {
    varName = Sr_GetVariableName(srObj);
    if(!F_StrIsEmpty(varName) {
        F_Printf(NULL, (StringT) "The variable name is %s\n",
                    varName);
        if (F_StrIEqual(varName, (StringT) "Current Date (Long)"))
        {
            parentElem = Sr_GetCurConvObjOfType(SR_OBJ_ELEM);
            parentEvent = Sr_GetAssociatedEvent(parentElem);
            if (F_StrIEqual(parentEvent->u.tag.gi, (StringT) "CPYRT"))
                Sr_SetVariableName(srObj,
                                    (StringT) "Current Date (Short)");
        }
        F_ApiDeallocateString(&varName);
    }
}
Sr_Convert(eventp, srObj);
}
```

See also

- [“Sr_SetVariableName\(\)” on page 235](#)
- [“SrConvObjT” on page 415](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_RowInUse()

Tests whether a specified row in a table is marked as used.

Synopsis

```
#include "fm_struct.h"
. . .
BoolT Sr_RowInUse(F_ObjHandleT docId, F_ObjHandleT rowId);
```

Arguments

<i>docId</i>	ID of the document containing the table
<i>rowId</i>	ID of the row in the table

Details

To test whether a row in a table is marked as used in the import process, call `Sr_RowInUse()`. Unused rows are deleted from a newly created table as the final step in the import process.

To mark a row as used, call `Sr_SetTableRowUsed()`.

Returns

1 if the row is marked as used; otherwise, 0. If *rowId* is not a row object, `SRW_errno` is set to `SRW_E_FAILURE`, and the function returns 0.

Examples

The following code tests the first row in a selected table to see if it is marked as used:

```
. . .
F_ObjHandleT docId, rowId, tableId;
BoolT rowIsUsed;

. . .
/* Get the document and table IDs. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tableId = F_ApiGetId(FV_SessionId, docId, FP_SelectedTbl);
/* Get the ID for the first row in the table. */
rowId = F_ApiGetId(docId, tblId, FP_FirstRowInTbl);

. . .
/* Test the first row to see if it is marked in use. */
rowIsUsed = Sr_RowInUse(docId, rowId);

. . .
```

See also

- [“Sr_SetTableRowUsed\(\)” on page 227](#)
- [“Sr_CellInUse\(\)” on page 103](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `F_ObjHandleT` and `BoolT`

Sr_SetAttrVal()

Sets the attribute value for a named attribute associated with the current import conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

SrwErrorT Sr_SetAttrVal(SrConvObjT srObj, F_AttributeT *attVal);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>attVal</i>	Pointer to the attribute-value data structure to use

Details

This function sets the passed attribute value structure to the specified attribute. Setting an attribute value is only valid if:

- The specified conversion object is associated with an event of type `SR_EVT_BEGIN_ELEM`.
- The import template's EDD includes an appropriate attribute definition for the proposed FrameMaker element.

You can use this function to set an attribute value to a FrameMaker element, even if the imported element has no attributes or attribute values. However, the EDD must define an appropriate attribute for the proposed element. The attribute name you provide must match the name of the FrameMaker attribute. For more information about how FrameMaker translates attributes, see *FrameMaker Structure Application Developer's Guide*.

This function is passed a structure of type `F_AttributeT`, which is defined as:

```
typedef struct (
    StringT name;
    F_StringsT values;
    UByteT valflags; /* validation error flag */
    UByteT allow; /* allow validation error as special case */
} F_AttributeT;
```

Note that `values` is a list of strings. Attribute values in markup can be character data, or they can be tokens or lists of tokens. If the value corresponds to character data or a single token, then you should set only one element in the list of strings. If the value corresponds to a list of tokens, then each string in the list corresponds to a single value token.

In FrameMaker, attributes are commonly used to affect the formatting of an element. For example, a list might have a `LIST_TYPE` attribute with possible values of `BULLET` or `NUM`,

which determines whether the list is bulleted or numbered. Your application can set the value of such an attribute depending on various criteria.

To set an attribute value, you must call this function before the object is converted.

Important: When you no longer need the attribute-value data structure, call `F_ApiDeAllocateAttribute()` to delete the structure and free the memory allocated for it.

To get an attribute value associated with a conversion object, call `Sr_GetAttrVal()`.

Returns

On error, one of the following values:

- `SRW_E_BAD_VALUE`
- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_NO_SUCH_ATTR`

Examples

The following code retrieves the attribute-value data structure for a `List_type` attribute associated with the current conversion object. Note that because FrameMaker attribute names are case sensitive, the code checks for the attribute name that would appear in the FrameMaker document, not the markup attribute name. The code then changes any `Num` attributes to `Bullet`. After changing the attribute value structure, it sets the structure to the current conversion object. Notice that the code frees the originally proposed attribute value before assigning a different string to it.

```
. . .
SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    F_AttributeT curAttr;
    SrEventT parentEventp;
    SrConvObjT listParentObj;
```

```
if (eventp->evtype == SR_EVT_BEGIN_ELEM) {
    if (F_StrIEqual(eventp->u.tag.gi, (StringT) "LIST")) {
        curAttr = Sr_GetAttrVal(srObj, (StringT) "List_type");
        if (F_StrIEqual(curAttr.values.val[0], (StringT) "NUM")) {
            F_Free(curAttr.values.val[0]);
            curAttr->values.val[0] =
                F_StrCopyString((StringT) "Bullet");
            Sr_SetAttrVal(srObj, &curAttr);
        }
        Sr_SetAttrVal(srObj, &curAttr);
        F_ApiDeallocateAttribute(&curAttr);
    }
}
. . .
}
```

See also

- [“Sr_GetAttrVal\(\)” on page 111](#)
- [“Sr_GetAttrVals\(\)” on page 114](#)
- [“Structured DeallocateAttrVals\(\)” on page 56](#)
- [“F_AttributeT” on page 410](#)
- [“SrConvObjT” on page 415](#)
- [“SrwErrorT” on page 396](#)

Sr_SetAttrVals()

Specifies the list of attribute values to use with the current conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_SetAttrVals(SrConvObjT srObj,
    F_AttributesT *attValList);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>attValList</i>	Pointer to the attribute-value list to use

Details

This function passes the list of attribute values to use with the current conversion object. The object can only have attribute values if:

- The current conversion object is associated with an event of type `SR_EVT_BEGIN_ELEM`.
- The import template's EDD includes appropriate attribute definitions for the proposed FrameMaker element.

You can use this function to set a list of attribute values to a FrameMaker element, even if the imported element has no attributes or attribute values. However, the EDD must define a appropriate attributes for the proposed element. The attribute names you provide must match the names of the FrameMaker attributes. For more information about how FrameMaker translates attributes, see *FrameMaker Structure Application Developer's Guide*.

This function is passed a structure of type `F_AttributesT`, which is defined as:

```
typedef struct {
    UIntT len; /* number of attributes */
    F_AttributeT *val; /* attributes array pointer */
} F_AttributesT;
```

`val` points to an array of attribute values, each one represented by a structure of type `F_AttributeT`, which is defined as:

```
typedef struct (
    StringT name;
    F_StringsT values;
    UByteT valflags; /* validation error flag */
    UByteT allow; /* allow validation error as special case */
} F_AttributeT;
```

Note that `values` is a list of strings. Attribute values in markup can be character data, or they can be tokens or lists of tokens. If the value corresponds to character data or a single token, then you should set only one element in the list of strings. If the value corresponds to a list of tokens, then each string in the list corresponds to a single value token.

For more information about single attribute values, see [“Sr_SetAttrVal\(\)” on page 184](#).

To set a list of attribute values, you must call this function before the object is converted.

Important: When you no longer need the `F_AttributesT` data structure, call `F_ApiDeallocateAttributes()` to delete the structure and free the memory allocated for it.

Returns

On error, one of the following values:

- SRW_E_BAD_VALUE
- SRW_E_WRONG_OBJ_TYPE

Examples

The following code retrieves the list of attribute values associated with the current conversion object. It uses two nested loops to look at all the tokens that make up the values for each attribute in the list. It then changes every attribute value of 1999 to 2000. After changing the attribute values structure, it sets the structure to the current conversion object.

```
. . .
SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    F_AttributesT curAttrVals;
    UIntT i, j;

    if (eventp->evtype == SR_EVT_BEGIN_ELEM) {
        curAttrVals = Sr_GetAttrVals(srObj);
        if(curAttrVals.len) {
            /* Loop through the list of attribute values */
            for (i=0; i < curAttrVals.len; i++) {
                /* Loop through attrVal string list */
                for (j = 0; j < curAttrVals.val[i].values.len; j++) {
                    if (F_StrIEqual(curAttrVals.val[i].values.val[j],
                                    (StringT) "1999")) {
                        F_Free(curAttrVals.val[i].values.val[j]);
                        curAttrVals.val[i].values.val[j] =
                            F_StrCopyString((StringT) "2000");
                    }
                } /* End of loop through curAttrVals.val[i]*/
            } /* End of loop through curAttrVals */
            Sr_SetAttrVals(srObj, &curAttrVals);
        } /* End of IF curAttrVals.len */
        F_ApiDeallocateAttributes(&curAttrVals);
    } /* End if IF ELEMENT */
    return (Sr_Convert(eventp, srObj));
} /* End of event */
```

See also

- [“Sr_GetAttrVals\(\)” on page 114](#)

- [“Sr_GetAttrVal\(\)” on page 111](#)
- [“Structured_DeallocateAttrVals\(\)” on page 56](#)
- [“F_AttributeT” on page 410](#)
- [“SrConvObjT” on page 415](#)
- [“SrwErrorT” on page 396](#)

Sr_SetBookCompFilePath()

Specifies the import filepath for the FrameMaker file in a book associated with an import conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_SetBookCompFilePath(SrConvObjT srObj,
    FilePathT *filePath);
```

Arguments

<i>srObj</i>	Conversion object to query
<i>filePath</i>	Pointer to the filepath to use

Details

Call `Sr_SetBookCompFilePath()` before converting an import conversion object of type `SR_OBJ_BOOK_COMP` to change the proposed import filepath for a FrameMaker book component. The filepath associated with this conversion object is used to create a new book component document.

`FilePathT` is a pointer to a platform-independent representation of a platform-specific path. For more information about working with platform-independent representations of filepaths, see the *FDK Programmer's Reference*. Use `F_FilePathFree()` to free the filepath when you have done with it.

To retrieve a book component filepath for a conversion object, call `Sr_GetBookCompFilePath()`.

Returns

On error, one of the following error codes:

- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_NOT_BOOK_COMP`

Examples

The following code sets the book component filepath for a conversion object:

```
. . .
SrErrorT errorStatus;
FilePathT *bookCompFp;
. . .
errorStatus = Sr_SetBookCompFilePath(SrObj, bookCompFp);
. . .
```

See also

- [“Sr_GetBookCompFilePath\(\)” on page 116](#)
- [“Sr_GetBookFilePath\(\)” on page 117](#)
- [“SrConvObjT” on page 415](#)
- [“FilePathT” on page 409](#)
- [“SrErrorT” on page 396](#)

Sr_SetBookFilePath()

Sets the platform-independent filepath of a book file for a specified import conversion object of type `SR_OBJ_BOOK`.

Synopsis

```
#include "fm_struct.h"
. . .
SrErrorT Sr_SetBookFilePath (SrConvObjT srObj,
                             FilePathT *filePathp);
```

Arguments

<i>srObj</i>	Conversion object for which to set a filepath
<i>filePathp</i>	Pointer to the filepath to use

Details

To change the book filepath for a specified import conversion object of type `SR_OBJ_BOOK`, call `Sr_SetBookFilePath()` before performing the conversion. The filepath associated with this conversion object is used to create a new FrameMaker book file.

filePathp is a pointer to a `FilePathT` structure containing a pointer to a platform-independent representation of a platform-specific path. For more information about working with platform-independent representations of filepaths, see the *FDK Programmer's Reference*. Use `F_FilePathFree()` to free the filepath when you have done with it.

To retrieve the proposed book filepath for a conversion object, call `Sr_GetBookFilePath()`.

Returns

On error, one of the following error codes:

- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_NOT_BOOK_COMP`

Examples

The following code retrieves a pointer to the filepath of a book file. If the pointer is `NULL`, it prompts for a valid filepath, and sets it before continuing.

```
. . .
FilePathT *bookFilePath
StringT fileName;
PathEnumT pform;
. . .
if ((bookFilePath = Sr_GetBookFilePath(srObj)) == NULL)
{
    PromptForBookFilePath(fileName, pform);
    bookFilePath = F_PathNameToFilePath(fileName, NULL, pform);
    Sr_SetBookFilePath(srObj, bookFilePath);
}
. . .
```

Ordinarily, FrameMaker always proposes a filepath for an `SR_OBJ_BOOK` conversion object. This example is for illustration only.

See also

- [“Sr_GetBookFilePath\(\)” on page 117](#)
- [“Sr_SetBookId\(\)” on page 192](#)
- [“SrConvObjT” on page 415](#)
- [“FilePathT” on page 409](#)
- [“SRW_errno” on page 418](#)

Sr_SetBookId()

Sets the ID for the FrameMaker book file that is associated with the specified import conversion object. Note that the conversion object must be one of type `SR_OBJ_BOOK`.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_SetBookId(SrConvObjT srObj, F_ObjHandleT bookId);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>bookId</i>	Book ID to assign to the object

Details

Note that you can't set a book ID until an event of type `SR_EVT_BEGIN_BOOK` has occurred along with the corresponding conversion object of type `SR_OBJ_BOOK`. A normal conversion using `Sr_Convert()` would result in creating a FrameMaker book, and the conversion object would keep that book's ID.

The idea of setting a book ID is to trap this import event before converting the object, open a book of your own design, and set that book's ID to the conversion object. Then all child objects (notably, book components) will be contained by the book you chose. This is a way to set up and use a book file with generated files in a specific order. It may be easier to do that once and save the file, rather than setting up the book programatically every time you import the document.

To change the ID of the FrameMaker book associated with current import conversion object, you should do the following:

- Ensure the current conversion object is type `SR_OBJ_BOOK`. Otherwise, this function returns an error of `SRW_E_INVALID_CONV_OBJ`.
- Create or open the alternate book you want to use, save it to the correct name and location, then get it's ID.
- Set the alternate book ID to the current conversion object of type `SR_OBJ_BOOK`.
- Call `Sr_Convert()` to complete the conversion of the book object.
- Convert the subsequent conversion objects normally. All subsequent conversion objects of type `SR_OBJ_BOOK_COMP`, and their child conversion objects, will be imported into the newly specified book.

Important: Be sure to call `Sr_SetBookId()` before the conversion object is converted. After you set the book ID, you should then call `Sr_Convert()` to convert this object. Using `Sr_Convert()` ensures a complete and clean conversion of the object, sharing needed book information with the rest of the import session.

Returns

On error, one of the following error codes:

- `SRW_E_INVALID_CONV_OBJ`
- `SRW_E_BAD_OBJ_HANDLE`

Examples

If the doctype is `SPECIAL`, the code opens the alternate book file, returns that book's ID, and sets the alternate book Id. All children of the book conversion object will be imported into the alternate book.

```
. . .
SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    F_ObjHandleT altBkId;

    if ((eventp->evtype) == SR_EVT_BEGIN_BOOK) {
        if (F_StrIEqual(Structured_GetDocTypeName(),
(StringT)"SPECIAL")) {
            /*Open the alternate book and get its ID*/
            altBkId = F_ApiSimpleOpen((StringT)
                                     "/mypath/mybook", False);
            Sr_SetBookId(srObj, altBkId);
        }
    }
    /*Now convert all objects normally*/
    Sr_Convert(eventp, srObj);
}
```

See also

- [“Sr_GetBookId\(\)” on page 119](#)
- [“Sr_SetDocId\(\)” on page 197](#)
- [“Sr_SetFlowId\(\)” on page 199](#)
- [“SrConvObjT” on page 415](#)

- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `F_ObjHandleT`

Sr_SetCharFmt()

Sets the character format tag to associate with the specified conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_SetCharFmt(SrConvObjT srObj, StringT charTag);
```

Arguments

srObj Conversion object to change

charTag Character format tag to set

Details

To set the character format tag to associate with a FrameMaker `SR_OBJ_SPECIAL_CHAR` conversion object, call `Sr_SetCharFmt()` before performing the conversion.

FrameMaker generates a `SR_OBJ_SPECIAL_CHAR` conversion object when importing an SDATA entity that has a corresponding read/write rule to convert that entity to a special FrameMaker character. There are specific issues associated with special characters. For example, the character format for any special character must specify a non-text font such as Symbol. (For more information see the online manual, *FrameMaker Structure Application Developer's Guide*.)

Call `Sr_GetCharFmt()` to retrieve the proposed character format tag name used by a special character conversion object.

Returns

On error, `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code sets the character format tag for every special character conversion object:

```
. . .
SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
```

```
{
    if (eventp->evtype == SR_EVT_BEGIN_ENTITY) {
        if (Sr_GetObjType(srObj) == SR_OBJ_SPECIAL_CHAR)
            Sr_SetCharFmt(srObj, (StringT)"MySymbols");
    }
    return (Sr_Convert(eventp, srObj));
}
```

See also

- [“Sr_GetCharFmt\(\)” on page 120](#)
- [“Sr_SetFmChar\(\)” on page 202](#)
- [“SrConvObjT” on page 415](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_SetColSpecs()

Specifies a list of *CALS* column specifications (*colspecs*) to associate with a specified table or table-part conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_SetColSpecs(SrConvObjT srObj,
    SrwColSpecsT *listp);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>listp</i>	Colspec list to assign

Details

To specify the list of *colspecs* to use for a table or table part (heading, body, or footing), call `Sr_SetColSpecs()` before converting the object. Use this function only for *CALS* tables. *srObj* must be one of the following types:

- `SR_OBJ_TABLE`
- `SR_OBJ_TABLE_HEADING`
- `SR_OBJ_TABLE_BODY`
- `SR_OBJ_TABLE_FOOTING`

To assign a list of specifications for *CALS* span specifications (*spanspecs*), call `Sr_SetSpanSpecs()`.

Returns

On error, `SRW_W_WRONG_OBJ_TYPE`.

Examples

The following code, from an event handler for a *CALS* table object, retrieves a *colspec* list, modifies it, and later assigns the modified list to the table object:

```
. . .
SrwErrorT Sr_EventHandler(SrEventT *eventp, SrConvObjT srObj)
{
    SrwColSpecsT colspecs;
    SrwColSpecT colspec;
    SrConvObjT tblObj;
    switch(Sr_GetObjType(srObj))
    {
        case SR_OBJ_TABLE_ROW:
            /* Retrieve the current table object. */
            tblObj = Sr_GetCurConvObjOfType(SR_OBJ_TABLE);
            /* Retrieve copy of colspecs for table object. */
            colspecs = Sr_GetColSpecs(tblObj);
            . . .
            /* Retrieve copy of colspec for the first column. */
            colspec = Srw_GetColSpecByColNum(&colspecs, 0);
            if(SRW_errno == SRW_E_SUCCESS)
            {
                /* Set the column width to 12 cm. */
                F_Free(colspec.width);
                colspec.width = F_StrCopyString("12cm");
                colspec.valueSet |= SRW_COLSPEC_COLWIDTH;
                /* Put new colspec back into list. */
                Srw_SetColSpec(&colspecs, &colspec);
                /* Put the new list back into the object. */
                Sr_SetColSpecs(tblObj, &colspecs);
            }
            /* Free colspec and colspecs. */
            Srw_DeallocateColSpec(&colspec);
            Srw_DeallocateColSpecs(&colspecs);
            break;
            . . .
    }
```

See also

- [“Sr_GetColSpecs\(\)” on page 124](#)

- [“Sr_SetSpanSpecs\(\)” on page 224](#)
- [“SrConvObjT” on page 415](#)
- [“SrwColSpecsT” on page 419](#)
- [“SrwErrorT” on page 396](#)

Sr_SetDocId()

Sets the document ID for the FrameMaker document associated with the specified import conversion object. Note that the conversion object must be one of type `SR_OBJ_DOC`.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_SetDocId(SrConvObjT srObj, F_ObjHandleT docId);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>docId</i>	Document ID to assign to the object

Details

Note that you can't set a document ID until an event of type `SR_EVT_BEGIN_DOC` has occurred along with the corresponding conversion object of type `SR_OBJ_DOC`. A normal conversion using `Sr_Convert()` would result in the creation of a FrameMaker document, and the conversion object would store that document's ID. This document would use the import template that is specified in the current structure application.

The idea of setting a document ID is to trap this conversion before it occurs, open a document of your own design, and set that document's ID to the conversion object so all child objects will be contained by that document. For example, the current structure application might specify an import template that is proper for the body of your document, but not for an appendix. With this function you can use a specially designed document whenever you encounter an appendix element in markup.

To change the ID of the FrameMaker document associated with current import conversion object, you should do the following:

- Ensure the current conversion object is type `SR_OBJ_DOC`. Otherwise, this function returns an error of `SRW_E_INVALID_CONV_OBJ`.
- Create or open the alternate document you want to use, then get its ID.
- Set the alternate document ID to the current conversion object of type `SR_OBJ_DOC`.

- Call `Sr_Convert()` to complete the conversion of the document object. Among other things, this sets the conversion object's flow ID to the main flow of your alternate document.
- Convert the subsequent conversion objects normally. All subsequent child conversion objects will be imported into the newly specified document.

Important: Be sure to call `Sr_SetDocId()` before the conversion object is converted. After you set the document ID, you should then call `Sr_Convert()` to convert this object. Using `Sr_Convert()` ensures a complete and clean conversion of the document, sharing needed information with the rest of the import session.

Returns

On error, one of the following error codes:

- `SRW_E_INVALID_CONV_OBJ`
- `SRW_E_BAD_OBJ_HANDLE`

Examples

The following code checks for a doctype of APPENDIX. If so, it opens an alternate document file and returns that document's ID. Then the code sets the alternate document Id. All elements in the instance of the APPENDIX doctype will be imported into the main flow of the alternate document.

```
. . .
SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    F_ObjHandleT altDocId;
    StringT docType;

    if ((eventp->evtype) == SR_EVT_BEGIN_DOC) {
        docType = Structured_GetDocTypeName();
        if (F_StrIEqual(, docType, (StringT)"APPENDIX")) {
            /*Open the alternate document and get its ID*/
            altDocId = F_ApiSimpleNewDoc((StringT)
                                      "/mypath/mydoc", False);
            Sr_SetDocId(srObj, altDocId);
        }
        F_ApiDeallocateStr(&docType);
    }
    /*Now convert all objects normally*/
    return(Sr_Convert(eventp, srObj));
}
```

See also

- [“Sr_GetDocId\(\)” on page 132](#)
- [“Sr_SetFlowId\(\)” on page 199](#)
- [“SrConvObjT” on page 415](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `F_ObjHandleT`

Sr_SetFlowId()

Sets the flow ID for the FrameMaker document associated with the specified import conversion object. Note that the conversion object must be one of type `SR_OBJ_DOC`.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_SetFlowId(SrConvObjT srObj, F_ObjHandleT flowId);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>flowId</i>	Flow ID to assign to the object

Details

Note that you can't set a flow ID until a FrameMaker document has been created to contain it. Documents can be created in the following ways:

- Passing a conversion object of type `SR_OBJ_DOC` to `Sr_Convert()`. This creates the FrameMaker document, and automatically sets that document's ID to the conversion object.
- Trapping a conversion object of type `SR_OBJ_DOC` and directly creating a FrameMaker document, then setting that document's ID to the conversion object via `Sr_SetDocId()`.

The document you are working with might have more than one flow in it. However, FrameMaker imports document content into the main flow by default. If you want to use a different flow you must set that flow's ID to the current document's conversion object.

Usually, the main flow is the default flow for the current language (in the English language, flow 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. While this algorithm usually works, you might find that it is insufficient for a given template. In that case, you might want to set the flow ID before you import the highest level element in the document.

To position elements on import, FrameMaker uses the `SrInsertLocT` structure to specify the insert location. If the element is the highest level element in the flow, this structure indicates a flow ID to specify the insert location. As a result, flow ID is only relevant when you are importing an element that is valid at the highest level in a document. This makes importing two structured flows into one document possible, but tricky. To switch flows mid-stream, you would need to first establish the root elements in each flow. Then to insert an element you would need to set the insert location to a position after the last element in the flow you want.

To change the ID of the flow associated with the current document conversion object, you should do the following:

- Make sure you haven't imported any elements into the document yet. Generally, you should set the flow ID as soon as you have an event of type `SR_EVT_BEGIN_DOC`.
- Get the current conversion object of type `SR_OBJ_DOC`.
- Get the document ID from the conversion object.
- Get the ID of your alternate flow from the document. Presumably, your document has more than one Autoconnect flow, and FrameMaker assumes the wrong one is the main flow.
- Set the alternate flow ID to the document conversion object.
- Convert the subsequent conversion objects normally. All subsequent child conversion objects will be imported into the newly specified flow.

Returns

On error, one of the following error codes:

- `SRW_E_INVALID_CONV_OBJ`
- `SRW_E_BAD_OBJ_HANDLE`

Examples

The following code checks for a doctype named `TEST`, for which you want to use a special import template. The code sets the new document ID, and ensures the content will go to the correct flow by setting the flow ID.

Note that master pages usually include flows with the same name as flows on the body pages. The code ensures the flow is on a body page by going up the hierarchy of document objects to check what type of page owns the first text frame in the flow.

```
#include "fm_struct.h"

SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
```



```
{
    F_ObjHandleT docId, altFlowId, tmpId;
    StringT dType, flowName;

    if (eventp->evtype == SR_EVT_BEGIN_DOC) {
        dType = Structured_GetDocTypeName();
        if (F_StrIEqual(dType, (StringT)"TEST")) {
            /* Open alt doc and set it as import template */
            docId = F_ApiSimpleNewDoc((StringT)
                                     "/mypath/mydoc", False);
            Sr_SetDocId(srObj, docId);
            /* Now set the flow ID. First find the appropriate */
            /* flow, ensuring the flow is on a body page. */
            altFlowId = F_ApiGetId(FV_SessionId, docId,
                                   FP_FirstFlowInDoc);
            while (altFlowId) {
                flowName = F_ApiGetStr(docId, altFlowId, FP_Name);
                if(F_StrIEqual((StringT)"FLOWB", flowName)) {
                    tmpId = F_ApiGetId(docId, altFlowId,
                                       FP_FirstTextFrameInFlow);
                    tmpId = F_ApiGetId(docId, tmpId, FP_FrameParent);
                    tmpId = F_ApiGetId(docId, tmpId,
                                       FP_PageFramePage);
                    if (F_ApiGetObjectType(docId, tmpId)==
                                                                FO_BodyPage)
                        break;
                } /* End IF FlowB
                F_ApiDeallocateString(&flowName);
                altFlowId = F_ApiGetId(
                    docId, altFlowId, FP_NextFlowInDoc);
            } /* End of LOOP through flows */
            if (altFlowId)
                Sr_SetFlowId(srObj, altFlowId);
        } /* End of IF "test" DOCTYPE */
        F_ApiDeallocateStr(&dType);
    } /* End of IF BEGIN DOC EVENT */
    return(Sr_Convert(eventp, srObj));
}
```

See also

- [“Sr_GetFlowId\(\)” on page 135](#)
- [“Sr_SetDocId\(\)” on page 197](#)
- [“SrConvObjT” on page 415](#)

- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `F_ObjHandleT`

Sr_SetFmChar()

Sets the FrameMaker special character assigned to an import conversion object of type `SR_OBJ_SPECIAL_CHAR`.

Synopsis

```
#include "fm_struct.h"

. . .

SrwErrorT Sr_SetFmChar(SrConvObjT srObj, PCharT fmChar);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>fmChar</i>	Character to set

Details

To map an entity to a different FrameMaker special character associated with an `SR_OBJ_SPECIAL_CHAR` import conversion object, call `Sr_SetFmChar()` before converting the object.

To retrieve the current special character associated with an import conversion object, call `Sr_GetFmChar()`.

Returns

On error, one of the following error codes:

- `SRW_E_BAD_VALUE`
- `SRW_E_WRONG_OBJ_TYPE`

Examples

The following code maps the special character for a conversion object:

```
. . .
SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
```

```
{
    if(eventp->evtype == SR_EVT_BEGIN_ENTITY) {
        if(Sr_GetObjType(srObj) == SR_OBJ_SPECIAL_CHAR) {
            Sr_SetFmChar(srObj, (UCharT)'\xd5');
            Sr_SetCharFmt(srObj, (StringT)"ARF");
        }
    }
    return(Sr_Convert(eventp, srObj));
}
```

See also

- [“Sr_GetFmChar\(\)” on page 137](#)
- [“Sr_SetFmChar\(\)” on page 202](#)
- [“SrConvObjT” on page 415](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on StringT for more information on PCharT

Sr_SetFmElemTag()

Specifies the tag of a FrameMaker element to be used by the specified conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_SetFmElemTag(SrConvObjT srObj, StringT fmTag);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>fmTag</i>	FrameMaker element tag to use

Details

This function sets the tag (name) of the FrameMaker element to be used by the specified conversion object. You might want to set an element tag if read/write rules prove insufficient for your conversion. For example, you might need two different elements for figures, one for body text and one for table content.

srObj must be a conversion object of the following types:

SR_OBJ_ELEM	SR_OBJ_TABLE	SR_OBJ_TABLE_TITLE
SR_OBJ_TABLE_HEADING	SR_OBJ_TABLE_BODY	SR_OBJ_TABLE_FOOTING
SR_OBJ_TABLE_ROW	SR_OBJ_TABLE_CELL	SR_OBJ_TABLE_COLSPEC

SR_OBJ_TABLE_SPANSPEC	SR_OBJ_GRAPHIC	SR_OBJ_EQUATION
SR_OBJ_VARIABLE	SR_OBJ_MARKER	SR_OBJ_XREF
SR_OBJ_FOOTNOTE		

To set an element tag, you must call this function before the object is converted.

To retrieve the proposed element tag for a conversion object, call `Sr_GetFmElemTag()`.

Returns

On error, `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code checks for a proposed element named TEST1 and changes the name for the FrameMaker element to "Tag2." When testing the string value, note that `F_StrIEqual()` is not case sensitive. However, when assigning the new element name, FrameMaker expects the case to be correct.

```
#include "fm_struct.h"
#include "fstrings.h"

SrwErrorT Sr_EventHandler(eventp, srObj)
SrConvObjT srObj;
{
    SrEventT *elemEvtp;
    StringT fmElemGi;

    if (eventp->evtype == SR_EVT_BEGIN_ELEM) {
        fmElemGi = Sr_GetFmElemTag(srObj);
        if (F_StrIEqual(fmElemGi, (StringT) "TEST1"))
            Sr_SetFmElemTag(srObj, (StringT) "Test2");
        F_ApiDeallocateString(&fmElemGi);
    }
    return (Sr_Convert(eventp, srObj));
}
```

See also

- [“Sr_GetFmElemTag\(\)” on page 140](#)
- [“Sr_UseFmElemId\(\)” on page 236](#)
- [“SrConvObjT” on page 415](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_SetFmText()

Assigns text to a specified conversion object of type `SR_OBJ_TEXT`.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_SetFmText(SrConvObjT srObj, StringT fmText);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>fmText</i>	Text to use

Details

FrameMaker imports CDATA and PCDATA via conversion objects of type `SR_OBJ_TEXT`. Conversion objects of this type are opened for every event of type `SR_EVT_CDATA`. Note that more than one such event can be posted for a single entity reference, but no more than one event is posted for a single character reference.

To specify the text that will be imported, you must call this function before the object is converted.

To retrieve the proposed FrameMaker text for a conversion object of type `SR_OBJ_TEXT`, call `Sr_GetFmText()`.

Returns

On error, `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code traps CDATA or PCDATA and strips leading spaces out of the text associated with the conversion object:

```
#include "fm_struct.h"
#include "fstrings.h"

SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    StringT s;
    s = F_StrNew((UIntT)256);
```

```
    if (eventp->evtype == SR_EVT_CDATA &&
        Sr_GetObjType(srObj) == SR_OBJ_TEXT) {
        s = Sr_GetFmText(srObj);
        F_StrStripLeadingSpaces(s);
        Sr_SetFmText(srObj, s);
        F_ApiDeallocateString(&s);
    }
    return(Sr_Convert(eventp, srObj));
}
```

See also

- [“Sr_GetFmText\(\)” on page 143](#)
- [“SrConvObjT” on page 415](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_SetImportFileHint()

Assigns a string identifying the file format and filter to use to import a graphic or equation associated with an import conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_SetImportFileHint(SrConvObjT srObj,
    StringT fileHint);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>fileHint</i>	String with file format and filter information

Details

To assign a string identifying the file format and filter to use to import a graphic or equation associated with an `SR_OBJ_GRAPHIC` or `SR_OBJ_EQUATION` conversion object, call `Sr_SetImportFileHint()` before converting the object.

The format for the *fileHint* string must either be `NULL` or take the form:

```
<record_ver><vendor><format_id><platform><filter_ver><filter_name>
>
```

Each field in the string except *filter_name* must be a 4-byte alphanumeric code. If a field code is less than 4 bytes, the extra bytes are padded with spaces. *filter_name* can be up to 31 alphanumeric characters in length. For example, a valid format could look like this:

```
0001PGRFPICTMAC61.0 Built-in PICT Writer
```

record_ver specifies the version number of the file format and filter string. It should be set to 0001.

vendor is a code specifying the filter's vendor. The code must always be 4 bytes, padded at the end with blanks if necessary. The following table lists valid vendor codes.

Code	Meaning
FRAM	Built-in FrameMaker filters
FAPI	External FrameMaker FDK client filters
FFLT	External FrameMaker filters
IMAG	External INSO graphic filters
XTND	Extended XTND filters

This table is not comprehensive. Codes may be added to this list by FrameMaker or by application developers at your site.

format_ID is a code specifying the file format that the filter translates. The code must always be 4 bytes, padded at the end with blanks if necessary. The following table lists many possible *format_ID* codes.

Code	Meaning
CDR	CorelDraw!
CGM	Computer Graphics Metafile
DIB	Device-independent bitmap (Windows)
DRW	Micrografx CAD
DXF	Autodesk Drawing eXchange file (CAD files)
EMF	Enhanced Metafile (Windows)
EPSB	Encapsulated PostScript® Binary (Windows)
EPSD	Encapsulated PostScript with Desktop Control Separations (DCS)
EPSF	Encapsulated PostScript (Macintosh)
EPSI	Encapsulated PostScript Interchange
FRMI	FramedImage
FRMV	FrameVector
G4IM	CCITT Group 4 to Image
GEM	GEM file (Windows)

Code	Meaning
GIF	Graphics Interchange Format (CompuServe)
HPGL	Hewlett-Packard Graphics Language
IGES	Initial Graphics Exchange Specification (CAD files)
IMG4	Image to CCITT Group 4 (Unix)
OLE	Object Linking and Embedding Client (Microsoft)
PCX	PC Paintbrush
PICT	Quickdraw PICT
PNTG	MacPaint
SNFR	Sun Raster File
SRGB	SGI RGB
TIFF	Tagged Image File Format
WMF	Windows Metafile
WPG	WordPerfect Graphics
XWD	X Windows System Window Dump file

platform is a 4-byte code specifying the platform on which the filter is run. The following table lists possible codes; it is also possible to use a code of four spaces.

Code	Meaning
MAC6	Macintosh 68000 series
MACP	Power Macintosh
OS/2	IBM OS/2
UNIX	Generic X/11 (Sun, HP)
WIN3	Windows 3.1
WIN4	Windows 95
WINT	Windows NT

filter_ver is a string of four characters identifying the version of the filter on the specified 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.

If you pass a `NULL` string, then on import, FrameMaker attempts to recognize the file format on its own and choose an existing filter to use for import.

You can retrieve the currently assigned file format and filter for import with `Sr_GetImportFileHint()`. When FrameMaker creates an initial `SR_OBJ_GRAPHIC` or `SR_OBJ_EQUATION` conversion object, it sets the file format and filter to `NULL`.

Returns

On error, one of the following error codes:

- `SRW_E_FAILURE`
- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_INVALID_CONV_OBJ`

Examples

The following code sets the file hint string for a conversion object:

```
. . .
SrErrorT errorStatus;
StringT fileHint;
. . .
errorStatus = Sr_SetImportFileHint(srObj, fileHint);
. . .
```

See also

- [“Sr_GetImportFileHint\(\)” on page 145](#)
- [“SrConvObjT” on page 415](#)
- [“SrErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_SetInsertedTablePartElementName()

Specifies the element name for a table or table part inserted into a FrameMaker document by a conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
VoidT Sr_SetInsertedTablePartElementName(SrConvObjT srObj,
SrTablePartTypeT tablePart, StringT elemName);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>tablePart</i>	Type of table part for which to set an element name
<i>elemName</i>	Element name to use

Details

To specify the name of a FrameMaker element for an inserted table or table part (heading, body, or footing), call `Sr_SetInsertedTablePartElementName()`.

Call `Sr_GetInsertedTablePartElementName()` to retrieve the element name for a table or table part inserted into a FrameMaker document by a conversion object.

Returns

VoidT.

Examples

The following code sets the element name for the body of a table:

```
. . .
StringT tblPrtElemName;
. . .
Sr_SetInsertedTablePartElementName (srObj, SRW_TABLE_BODY,
    tblPrtElemName);
. . .
```

See also

- [“Sr_GetInsertedTablePartElementName\(\)” on page 146](#)
- [“SrConvObjT” on page 415](#)
- [“SrwTablePartTypeT” on page 403](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_SetInsertLoc()

Sets the insertion location in the FrameMaker document for the FrameMaker object corresponding to a specific conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_SetInsertLoc(SrConvObjT srObj,
    SrInsertLocT *insertLoc);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>insertLoc</i>	Pointer to structure containing insert location information

Details

This function specifies where in the current FrameMaker document the specified conversion object's content will be inserted. This location corresponds to an insertion point in the document. You typically use this function when the order of elements in the FrameMaker document must be different than the order in the markup.

Note that you must call this function before the conversion object is converted.

The insert location is specified in a structure of type `SrInsertLocT`, which is defined as:

```
typedef struct {
    SrLocationT pos;
    union {
        F_ObjHandleT flowId;
        F_ObjHandleT mkrId;
        F_ElementLocT elemLoc;
    } u;
} SrInsertLocT;
```

For more information about `SrInsertLocT`, see the *FDK Programmer's Reference* and the *FDK Programmer's Guide*.

The possible values for `pos` are:

- `SR_LOC_FLOW`

`u` contains a flow ID that indicates the flow into which the first element should be inserted. Note that this is only valid when the proposed element is valid at the highest level in the flow.

- `SR_LOC_MARKER_TEXT`

`u` contains a marker ID that indicates the FrameMaker marker into which text should be inserted.

- `SR_LOC_ELEMENT`:

`u` contains an element location that indicates the location in the document where the new text, element, or object should be inserted. `F_ElementLocT` is defined as:

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

The parent element is the element that will contain the inserted element. The child element is the sibling of the inserted element that immediately follows the inserted element. If `child` is 0, the inserted element will be the last child of the parent.

The offset counts at what number of characters into the parent to insert the current element. (Note that an offset can also count into a child element, but only when dealing with a selection range. For this function, selection ranges are never the case.)

- SR_LOC_BOOK

u is ignored, and the element is inserted into the current book.

You can inspect a proposed insert location with `Sr_GetInsertLoc()`.

Returns

On error, one of the following error codes:

- SRW_E_BAD_OBJ_HANDLE
- SRW_E_BAD_VALUE

Examples

For an example that shows how to re-order elements in both import and export events, see [“Sw_ScanElem\(\)” on page 340](#).

The following example shows how to re-order elements when you import markup. It ensures the proposed insert location for a GLOSSARY conversion object is the first element position in its parent element. For example, if ENDMATTER contains ENDNOTES and GLOSSARY, the code imports GLOSSARY as the first child in ENDMATTER.

```
. . .
SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    if (F_StrIEqual(eventp->u.tag.gi, (StringT) "GLOSSARY")) {
        SrInsertLocT insLoc;
        SrConvObjT docObj;
        F_ObjHandleT firstChildId, docId;

        docObj = Sr_GetCurConvObjOfType(SR_OBJ_DOC);
        if (!docObj)
            docObj = Sr_GetCurConvObjOfType(SR_OBJ_BOOK_COMP);
        if (docObj)
            docId = Sr_GetDocId(docObj);

        /* Make this element is the first child of its parent */
        insLoc = Sr_GetInsertLoc(srObj);

        insLoc.u.elemLoc.childId = F_ApiGetId(docId,
            insLoc.u.elemLoc.parentId, FP_FirstChildElement);
        insLoc.u.elemLoc.offset = 0;

        Sr_SetInsertLoc(srObj, &insLoc);
    }
}
```

```
        return(Sr_Convert(eventp, srObj));  
    }  
    . . .
```

See also

- [“Sr_GetInsertLoc\(\)” on page 147](#)
- [“SrInsertLocT” on page 416](#)
- [“SrConvObjT” on page 415](#)

Sr_SetLineBrkInfo()

Sets the line-break information for the specified import conversion object.

Synopsis

```
#include "fm_struct.h"  
.  
.  
.  
SrwErrorT Sr_SetLineBrkInfo(SrConvObjT srObj, IntT lineBrkInfo);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>lineBrkInfo</i>	SR_LB_SPACE or SR_LB_FORCED_RETURN

Details

This function determines whether to translate each record end (RE) for an element’s content as a space or as a forced return.

The PCDATA that is contained by an element in markup can be typed into the file as many lines of text, and each new-line character in the PCDATA is recognized as an RE. By default, FrameMaker translates any such RE as a space character. In other words, all the lines of text in the markup element get inserted in the FrameMaker element as one stream of text. The line length of this text is determined by FrameMaker’s column width, and word-wrap properties.

Note that a forced return starts the text on a new line, but does not begin a new paragraph.

You should be aware that you can set up read/write rules to specify this line break info without writing an API client. This function provides a way to override the line break rule for the current element. You can check the line break setting via `Sr_GetLineBreakInfo()`.

To change the line break info, call this function before the object is converted.

Returns

On error, SRW_E_FAILURE.

Examples

The following code retrieves the proposed line-break information for an event of type `SR_EVT_BEGIN_ELEM`, and if the corresponding element is tagged `CODE_SEGMENT`, it forces a new line for every record end in the element's content.

```
#include "fm_struct.h"

SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    SrConvObjT elemObj;
    SrEventT *elemEvt;
    IntT linebreak = 0;

    if (eventp->evtype == SR_EVT_BEGIN_ELEM) {
        if (F_StrIEqual(eventp->u.tag.gi,
                        (StringT) "CODE_SEGMENT")) {
            linebreak = Sr_GetLineBrkInfo(srObj);
            if (linebreak != SR_LB_FORCED_RETURN)
                Sr_SetLineBrkInfo(srObj, SR_LB_FORCED_RETURN);
        }
        return (Sr_Convert(eventp, srObj));
    }
}
```

See also

- [“Sr_GetLineBrkInfo\(\)” on page 150](#)
- [“SrConvObjT” on page 415](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `IntT`

Sr_SetPrivateData()

Sets to the specified conversion object a pointer to data your application allocates, initializes, and maintains.

Synopsis

```
#include "fm_struct.h"

. . .

VoidT Sr_SetPrivateData(SrConvObjT srObj, PtrT privDatap);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>privDatap</i>	Pointer to data to insert as private data

Details

You can allocate and maintain data that is private to your application, and then assign the address of that data to a specific conversion object. This is a way to save a state within your application, and then associate it with a specific level in the hierarchy of the conversion object stack. This function sets a pointer to such data to the specified conversion object.

Note that your application owns the data. Also, the data can be of any addressable type, from a single variable to a structure within a deep hierarchy of structures. It is up to your application to maintain the data content.

Important: When the conversion object is closed, it is removed from the conversion object stack and its allocated memory is freed. However, the data referenced by the private data pointer is not deallocated. If the only reference to this data is stored with the conversion object, and that object is closed, you will not be able to deallocate the private data. Be sure to trap ending events (for example, `SR_EVT_END_ELEM`) and check their associated objects for private data pointers.

Returns

VoidT.

Examples

For an example us getting and setting private data, see [“Sr_GetPrivateData\(\)” on page 155](#)

See also

- [“Sr_GetPrivateData\(\)” on page 155](#)
- [“SrConvObjT” on page 415](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `PtrT`

Sr_SetProcessingFlags()

Sets the flags that indicate the processing for the specified import conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

SrwErrorT Sr_SetProcessingFlags(SrConvObjT srObj, IntT flags);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>flags</i>	Integer containing desired processing flags

Details

To specify the processing for a conversion object corresponding to a FrameMaker element, you must call `Sr_SetProcessingFlags()` before converting the object. The flags argument can be set to one of the following values:

- `NULL`

This is equivalent to no read/write rules.

- `SRW_UNWRAP_ELEMENT`

This is equivalent to the `unwrap` read/write rule.

- `SRW_DROP_ELEMENT_CONTENT`

This is equivalent to the `drop content` read/write rule.

- `SRW_UNWRAP_ELEMENT | SRW_DROP_ELEMENT_CONTENT`

This is equivalent to both the `unwrap` and `drop content` read/write rules for a single element.

The conversion object must correspond to a FrameMaker element. If you try to set the processing flags for a non element conversion object, this function returns an error.

Note that if you drop the content of an element, none of the child elements will post an event.

Returns

On error, one of following error codes:

- `SRW_E_INVALID_CONV_OBJ`
- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_BAD_OBJ_HANDLE`

Examples

For an example of getting and setting processing flags, see [“Sr_GetProcessingFlags\(\)” on page 158](#).

See also

- [“Sr_GetProcessingFlags\(\)” on page 158](#)
- [“SrConvObjT” on page 415](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `IntT`

Sr_SetPropVal()

Sets a specified FrameMaker property value for an import conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

SrwErrorT Sr_SetPropVal(SrConvObjT srObj,
    SrwFmPropertyT fmProp, StringT propVal);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>fmProp</i>	Property structure to modify
<i>propVal</i>	String containing the new property value

Details

Conversion objects can have properties associated with them. The properties that can be assigned to an object depend on the conversion object type. The objects that can have properties are of the following types:

SR_OBJ_ELEM	SR_OBJ_TABLE	SR_OBJ_TABLE_TITLE
SR_OBJ_TABLE_HEADING	SR_OBJ_TABLE_BODY	SR_OBJ_TABLE_FOOTING
SR_OBJ_TABLE_ROW	SR_OBJ_TABLE_CELL	SR_OBJ_TABLE_COLSPEC
SR_OBJ_TABLE_SPANSPEC	SR_OBJ_GRAPHIC	SR_OBJ_EQUATION
SR_OBJ_VARIABLE	SR_OBJ_MARKER	SR_OBJ_XREF
SR_OBJ_FOOTNOTE		

For a list of the properties for these conversion object types, see [“SrwFmPropertyT” on page 397](#).

Note that the property value is a string, in spite of the fact that some of the properties are enumerated types or other numeric values. These numeric values are handled as strings to unify the format of all conversion object properties.

Important: When you are finished with a string, call `F_StrFree()` to free its memory.

To set a property value, you must call this function before the specified object is converted. Once converted, a property value for the object cannot be changed.

To retrieve a single property value for an import conversion object, call `Sr_GetPropVal()`.

Returns

On error, one of the following error codes:

- `SRW_E_BAD_VALUE`
- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_OBJ_HAS_NO_SUCH_PROP`

Examples

For an example of getting and setting a property value, see [“Sr_GetPropVal\(\)” on page 159](#)

See also

- [“Sr_GetPropVal\(\)” on page 159](#)
- [“Sr_SetPropVals\(\)” on page 219](#)
- [“SrConvObjT” on page 415](#)
- [“SrwFmPropertyT” on page 397](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_SetPropVals()

Specifies a list of properties for the specified conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

SrwErrorT Sr_SetPropVals(SrConvObjT srObj,
    SrwPropValstT *propVals);
```

Arguments

srObj Conversion object to change

propVals Pointer to the property values list to use

Details

Conversion objects can have properties associated with them. The properties that can be assigned to an object depend on the conversion object type. The objects that can have properties are of the following types:

SR_OBJ_ELEM	SR_OBJ_TABLE	SR_OBJ_TABLE_TITLE
SR_OBJ_TABLE_HEADING	SR_OBJ_TABLE_BODY	SR_OBJ_TABLE_FOOTING
SR_OBJ_TABLE_ROW	SR_OBJ_TABLE_CELL	SR_OBJ_TABLE_COLSPEC
SR_OBJ_TABLE_SPANSPEC	SR_OBJ_GRAPHIC	SR_OBJ_EQUATION
SR_OBJ_VARIABLE	SR_OBJ_MARKER	SR_OBJ_XREF
SR_OBJ_FOOTNOTE		

For a list of these properties, see [“SrwFmPropertyT” on page 397](#).

This function is passed a `SrwPropValsT`, which is a list of all the properties for a specified conversion object. `SrwPropValsT` is defined as:

```
typedef struct {
    IntT len; /* Number of properties in array. */
    SrwPropValT *vals; /* Pointer to array of values. */
} SrwPropValsT;
```

Each property is expressed by a `SrwPropValT` structure, which is defined as:

```
typedef struct {
    SrwFmPropertyT prop;
    StringT value;
} SrwPropValT;
```

Note that the property value is a string, in spite of the fact that some of the properties are enumerated types or other numeric values. These numeric values are handled as strings to unify the format of all conversion object properties.

Important: Note that actual property values are strings. If you want to store a value in a variable, then you must either use `Sr_GetPropVal()`, or else use `F_StrCopyString()`, as follows:

```
propVal = F_StrCopyString(propVals.vals[i].value).
```

Important: When you are finished with the list of property values, be sure to call `Srw_DeallocatePropVals()` to free its memory. Use `Srw_DeallocatePropVal()` to free up a single property value.

To change the list of property values you must call this function before the specified object is converted. Once converted, the object's list of values cannot be changed.

To retrieve the current list of property values for a conversion object, call `Sr_GetPropVals()`. To set a single property value, call `Sr_SetPropVal()`.

Returns

On error, one of the following error codes:

- `SRW_E_BAD_VALUE`
- `SRW_E_WRONG_OBJ_TYPE`

Examples

For an example of getting and setting property values, see [“Sr_GetPropVals\(\)” on page 162](#).

See also

- [“Sr_GetPropVals\(\)” on page 162](#)
- [“Srw_DeallocatePropVals\(\)” on page 253](#)
- [“Sr_SetPropVal\(\)” on page 217](#)
- [“SrConvObjT” on page 415](#)
- [“SrwPropValsT” on page 421](#)
- [“SrwErrorT” on page 396](#)

Sr_SetRefElemTag()

Sets the name of the reference element to associate with the specified import conversion object of type `SR_OBJ_REF_ELEM`. Objects of this type are proposed when importing an SDATA entity, or when the read/write rules convert an entity to a reference element.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_SetRefElemTag(SrConvObjT srObj, StringT refTag);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>refTag</i>	Name of reference element to use

Details

In markup documents, SDATA entities are used to represent information that might be referenced a number of times in the markup document. These can require processing, such as a current date, or they can be static data such as copyright boilerplate or a graphic for the company logo. FrameMaker can handle such entities in various ways. The current date is best imported as a FrameMaker date variable. Likewise, simple boilerplate could be imported as a user-defined variable, or complex boilerplate might be imported as a text inset. These must be set up in the import template, and your structure application must include read/write rules to convert instances of the specific entity as the correct variable.

You can also import an SDATA entity as almost any set of document objects. For example, an entity can be imported as one or more anchored frames. To do this, you map an SDATA entity to an element that you store on a reference page of your import template. You then use read/write rules to map the entity to the reference element. For more information, see “Translating SDATA entities as FrameMaker reference elements” in the *FrameMaker Structure Application Developer's Guide*.

It is possible that you might want two or more reference elements for a given SDATA entity. For example, you might have a large company logo for instances within regular body text, and a smaller logo for instances within tables, each wrapped in differently named reference elements. You can only specify one of these logo elements via read/write rules. With `Sr_GetRefElemTag()` you can check to see what the proposed reference element is, and with `Sr_SetRefElemTag()` you can change that proposal.

You must call this function before the object is converted because you cannot change the reference element afterwards.

To determine the proposed name of a reference element associated with an import conversion object, but not yet converted, call `Sr_GetRefElemTag()`.

Returns

On error, one of the following error codes:

- `SRW_E_BAD_VALUE`
- `SRW_E_WRONG_OBJ_TYPE`

Examples

For an example of getting and setting a reference element, see [“Sr_GetRefElemTag\(\)” on page 165](#).

See also

- [“Sr_GetRefElemTag\(\)” on page 165](#)
- [“SrConvObjT” on page 415](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_SetSessionProps()

Specifies import session properties for the current session (as specified by a conversion object of type `SR_OBJ_SESSION`).

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_SetSessionProps(SrConvObjT srObj,
    SrSessionPropsT *sessionProps);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>sessionProps</i>	Pointer to the structure containing session property settings

Details

To set the current session properties, you pass a `SrSessionPropsT` structure and the current session conversion object. To get the current session object, call `Sr_GetCurConvObjOfType(SR_OBJ_SESSION)`.

SrSessionPropsT is defined as:

```
typedef struct {
    BoolT isBatchMode; /*True if the current session is a batch.
                        NOTE: This field is Read Only.*/
    StringT tableRulingStyle; /*Name of a table ruling style*/
    BoolT overwriteFiles; /*True if batch can overwrite files of
                          same name in the target directory*/
} SrSessionPropsT;
```

To retrieve the current import session properties, call `Sr_GetSessionProps()`.

Returns

On error, one of the following error codes:

- `SRW_E_BAD_OBJ_HANDLE`
- `SRW_E_INVALID_CONV_OBJ`

Examples

The following case statement assigns the overwrite and default table ruling properties to the current session. Finally, the code frees the space allocated to the `SrSessionPropsT` structure:

```
. . .
SrSessionPropsT importSessionProps;
SrConvObjT sessionObj;
. . .
case SR_EVT_BEGIN_READER:
    Sr_Convert(eventp, srObj);
    sessionObj = Sr_GetCurConvObjOfType(SR_OBJ_SESSION);
    importSessionProps = Sr_GetSessionProps(sessionObj);
    importSessionProps.overwriteFiles = True;

    Sr_SetSessionProps(sessionObj, &importSessionProps);
    Sr_DeallocateSessionProps(&importSessionProps);
    break;
. . .
```

See also

- [“Sr_GetSessionProps\(\)” on page 167](#)
- [“Sr_DeallocateSessionProps\(\)” on page 105](#)
- [“Sr_GetCurConvObjOfType\(\)” on page 129](#)
- [“SrConvObjT” on page 415](#)
- [“SrwErrorT” on page 396](#)

Sr_SetSpanSpecs()

Specifies a list of *CALS* span specifications (*spanspecs*) to associate with a specified table or table-part conversion object.

Synopsis

```
#include "fm_struct.h"

. . .
SrwErrorT Sr_SetSpanSpecs(SrConvObjT srObj,
    SrwSpanSpecsT *listp);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>listp</i>	SpanSpec list to assign

Details

To specify the list of *spanspecs* to use for a table or table part (heading, body, or footing), call *Sr_SetSpanSpecs()* before converting the object. *srObj* must be one of the following types:

- *SR_OBJ_TABLE*
- *SR_OBJ_TABLE_HEADING*
- *SR_OBJ_TABLE_BODY*
- *SR_OBJ_TABLE_FOOTING*

After you convert the object, call *Srw_DeallocateSpanSpecs()* to delete the list of spanspecs and free the memory allocated for it.

To assign a list of specifications for *CALS colspecs*, call *Sr_SetColSpecs()*.

Returns

On error, *SRW_E_WRONG_OBJ_TYPE*.

Examples

For an example of getting and setting span specs, see [“Sr_GetSpanSpecs\(\)” on page 169](#).

See also

- [“Sr_GetSpanSpecs\(\)” on page 169](#)
- [“Srw_DeallocateSpanSpecs\(\)” on page 255](#)
- [“SrConvObjT” on page 415](#)
- [“SrwSpanSpecsT” on page 423](#)
- [“SrwErrorT” on page 396](#)

Sr_SetStraddles()

Specifies a list of running straddles to associate with a specified import conversion object of type `SR_OBJ_TABLE` created with a `start vertical straddle read/write` rule.

Synopsis

```
#include "fm_struct.h"

. . .
SrwErrorT Sr_SetStraddles(SrConvObjT srObj,
    SrwStraddlesT *listp);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>listp</i>	Running straddles list to assign

Details

To specify a list of running straddles for an import conversion object of type `SR_OBJ_TABLE`, call `Sr_SetStraddles()` before converting the object. After you convert the object, call `Srw_DeallocateStraddles()` to delete the list of straddles and free the memory allocated for it.

To retrieve a list of running straddles for a table, call `Sr_GetStraddles()`.

Returns

On error, `SRW_W_WRONG_OBJ_TYPE`.

Examples

The following code, from an event handler for a table object, retrieves a running-straddles list, modifies it, and later assigns the modified list to the table object:

```
. . .
SrwErrorT Sr_EventHandler(SrEventT *eventp, SrConvObjT srObj)
{
    SrwStraddlesT straddles;
    SrConvObjT tblObj;
    switch(Sr_GetObjType(srObj))
    {
        case SR_OBJ_TABLE_ROW:
            /* Retrieve the current table object. */
            tblObj = Sr_GetCurConvObjOfType(SR_OBJ_TABLE);
            /* Retrieve copy of running straddles for table. */
            straddles = Sr_GetStraddles(tblObj);

. . .

            /* Modify something in the straddles list here. */

. . .
```

Sr_SetTableCellUsed()

```
        /* Put the new list back into the object. */
        Sr_SetStraddles(tblObj, &straddles);
    }
    Srw_DeallocateStraddles(&straddles);
    break;

    . . .
```

See also

- [“Sr_GetStraddles\(\)” on page 170](#)
- [“Srw_DeallocateStraddles\(\)” on page 257](#)
- [“SrConvObjT” on page 415](#)
- [“SrwStraddlesT” on page 424](#)
- [“SrwErrorT” on page 396](#)

Sr_SetTableCellUsed()

Sets a flag on a specified conversion object of type `SR_OBJ_TABLE_CELL` to indicate that the cell is used in the import process.

Synopsis

```
#include "fm_struct.h"

. . .
VoidT Sr_SetTableCellUsed(F_ObjHandleT docId,
    F_ObjHandleT cellId);
```

Arguments

<i>docId</i>	ID of the document containing the table
<i>cellId</i>	ID of the cell in the table

Details

To set a flag indicating that a specific cell in a table is used in the import process, call `Sr_SetTableCellUsed()`. FrameMaker tests this flag to determine where the current cell should be located in a table.

Returns

VoidT.

Examples

The following code sets the used flag for the first cell in the first row in a selected table:

```
. . .
F_ObjHandleT docId, cellId, rowId, tableId;
. . .
/* Get the document and table IDs. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tableId = F_ApiGetId(FV_SessionId, docId, FP_SelectedTbl);
/* Get the ID for the first row in the table. */
rowId = F_ApiGetId(docId, tableId, FP_FirstRowInTbl);
/* Get the ID for the first cell of the first row. */
cellId = F_ApiGetId(docId, rowId, FP_FirstCellInRow)
. . .
/* Set the used flag for the cell. */
Sr_SetTableCellUsed(docId, cellId);
. . .
```

See also

- [“Sr_CellInUse\(\)” on page 103](#)
- [“Sr_SetTableRowUsed\(\)” on page 227](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `F_ObjHandleT` and `VoidT`

Sr_SetTableRowUsed()

Sets a flag on a specified table row to indicate that the row is used in the import process.

Synopsis

```
#include "fm_struct.h"
. . .
VoidT Sr_SetTableRowUsed(F_ObjHandleT docId, F_ObjHandleT rowId);
```

Arguments

<i>docId</i>	ID of the document containing the table
<i>rowId</i>	ID of the row in the table

Details

To set a flag indicating that a specific row in a table is used in the import process, call `Sr_SetTableRowUsed()`. For example, by default, when FrameMaker creates a table, it includes header and footer rows. If the finished table does not use these rows, FrameMaker deletes them as a final step. If you use custom code to create a table, you can call `Sr_SetTableRowUsed()` to prevent FrameMaker from deleting these rows.

Rows for which this flag is not set are purged on import.

Returns

VoidT.

Examples

The following code sets the used flag for the first row in a selected table:

```
. . .
F_ObjHandleT docId, rowId, tableId;
. . .
/* Get the document and table IDs. */
docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
tableId = F_ApiGetId(FV_SessionId, docId, FP_SelectedTbl);
/* Get the ID for the first row in the table. */
rowId = F_ApiGetId(docId, tableId, FP_FirstRowInTbl);
. . .
/* Set the used flag for the first row. */
Sr_SetTableRowUsed(docId, rowId);
. . .
```

See also

- [“Sr_RowInUse\(\)” on page 182](#)
- [“Sr_AddTableRows\(\)” on page 99](#)
- [“Primitive data types” on page 383](#) for more information on StringT for more information on F_ObjHandleT and VoidT

Sr_SetTextInsetFilePath()

Specifies the filepath for a FrameMaker text inset associated with an import conversion object of type SR_OBJ_TEXT_INSET.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_SetTextInsetFilePath(SrConvObjT srObj
FilePathT *filePath);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>filePath</i>	Pointer to the filepath to set

Details

To specify the filepath for a FrameMaker text inset to associate with an `SR_OBJ_TEXT_INSET` conversion object, call `Sr_SetTextInsetFilePath()` before converting the object.

filePathp is a pointer to a `FilePathT` structure containing a pointer to a platform-independent representation of a platform-specific path. For more information about working with platform-independent representations of filepaths, see Chapter 16, “Making I/O and Memory Calls Portable,” in the *FDK Programmer’s Guide*.

You can determine the current text inset filepath for an import conversion object by calling `Sr_GetTextInsetFilePath()`.

Returns

On error, one of the following error codes:

- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_FAILURE`

Examples

The following changes the paths for text inset source files. If the sources were previously in a directory that was a descendent of the “Test” sub-directory, this code changes the reference to the same filename, but in `C:\Archive\App\`. Note that since not all entity events require text insets, you must test for the result of `Sr_GetTextInsetFilePath()` before you can assume the entity has a text inset file path.

```
. . .
#define ANCHOR (StringT) "<v>C:<c>Archive<c>App"
#define SUB_DIR (StringT) "\\Test\\"
#define NSUB_DIR (sizeof(SUB_DIR)-1)/* length of \Test\ */
. . .
FilePathT *textInsetPath;
if (textInsetPath = Sr_GetTextInsetFilePath(srObj)) {
    IntT k;
    UCharT *s = F_FilePathToPathName(textInsetPath,
                                     FDefaultPath);
    FilePathT *anchor =
        F_PathNameToFilePath(ANCHOR, NULL, FDIPath);
```

```
/*
 * Replace filenames like ...\\Test\\myfile.gif with
 * filenames like c:\\Archive\\App\\myfile.gif
 */
if ((k = F_StrSubString(s, SUB_DIR)) != -1
    && !F_StrChr(&s[k+NSUB_DIR], (StringT) "\\\"))
{
    UCharT file[256+sizeof("<c>")];
    FilePathT *altPath;
    F_Printf(0, "\\nFileName Is: %s", &s[k+NSUB_DIR]);
    F_Sprintf(file, "<c>%s", &s[k+NSUB_DIR]);
    altPath = F_PathNameToFilePath(file, anchor, FDIPath);
    if (altPath)
        Sr_SetTextInsetFilePath(srObj, altPath);
}
F_ApiDeallocateString(&s);
F_FilePathFree(textInsetPath);
F_FilePathFree(anchor);
}
```

. . .

See also

- [“Sr_GetTextInsetFilePath\(\)” on page 174](#)
- [“Sr_SetTextInsetFlowTag\(\)” on page 231](#)
- [“Sr_SetTextInsetPageSpace\(\)” on page 234](#)
- [“SrConvObjT” on page 415](#)
- [“FilePathT” on page 409](#)
- [“SrwErrorT” on page 396](#)

Sr_SetTextInsetFlowTag()

Specifies the name of the FrameMaker text flow for a text inset associated with an import conversion object of type `SR_OBJ_TEXT_INSET`.

Synopsis

```
#include "fm_struct.h"

. . .
SrwErrorT Sr_SetTextInsetFlowTag(SrConvObjT srObj,
StringT flowTag);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>flowTag</i>	Flow tag to assign

Details

To specify the name of a FrameMaker text flow for a text inset to associate with an `SR_OBJ_TEXT_INSET` conversion object, call `Sr_SetTextInsetFlowTag()` before converting the object. *flowTag* should be the name of an existing flow in the document.

To determine the currently proposed flow name for a text inset, call `Sr_GetTextInsetFlowTag()`.

Returns

On error, one of the following error codes:

- `SRW_E_BAD_VALUE`
- `SRW_E_WRONG_OBJ_TYPE`

Examples

The following code checks for any entity that imports as text inset referring to a FrameMaker document named “MyFile.fm.” When it finds that entity, the code makes sure the text inset reference is to the content of flow B.

```
. . .
FilePathT *textInsetPath;
. . .
```

Sr_SetTextInsetFormatting()

```
case SR_EVT_BEGIN_ENTITY:
    if(F_StrIEqual((StringT)"MyEnt", eventp->u.entname)) {
        textInsetPath = Sr_GetTextInsetFilePath(srObj);
        if(-1 != F_StrSubString((StringT)"MyFile.fm",
            F_FilePathToPathName(textInsetPath, FDefaultPath))) {
            Sr_SetTextInsetFlowTag(srObj, (StringT)"B");
        }
        F_FilePathFree(textInsetPath);
    }
    break;
. . .
```

See also

- [“Sr_GetTextInsetFlowTag\(\)” on page 176](#)
- [“Sr_SetTextInsetFilePath\(\)” on page 228](#)
- [“Sr_SetTextInsetPageSpace\(\)” on page 234](#)
- [“SrConvObjT” on page 415](#)
- [“SrWrErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_SetTextInsetFormatting()

Sets how the content is formatted in a text inset associated with an import conversion object of type `SR_OBJ_TEXT_INSET`.

Synopsis

```
#include "fm_struct.h"
. . .
IntT Sr_GetTextInsetFormatting(SrConvObjT srObj, IntT tiFmt);
```

Arguments

<i>srObj</i>	Conversion object to query
<i>tiFmt</i>	The type of formatting; one of <code>FV_SourceDoc</code> , <code>FV_EnclosingDoc</code> , or <code>FV_PlainText</code>

Details

Call `Sr_SetTextInsetFormatting()` to specify how the content of a text inset associated with an `SR_OBJ_TEXT_INSET` conversion object will be formatted. The content can be formatted according to the source document, according to the importing document, or formatted as plain text.

To determine the proposed formatting for an inset, call `Sr_GetTextInsetFormatting()`.

Returns

On error, one of the following error codes:

- `SRW_E_BAD_VALUE`
- `SRW_E_WRONG_OBJ_TYPE`

Examples

The following code retrieves the formatting proposed for the text inset associated with the current conversion object, and ensures it will be according the to enclosing document:

```
. . .
FilePathT *textInsetPath;
IntT fmt;
. . .
case SR_EVT_BEGIN_ENTITY:
    if(textInsetPath = Sr_GetTextInsetFilePath(srObj)) {
        fmt = Sr_GetTextInsetformatting(srObj);
        if(fmt == FV_EnclosingDoc)
            Sr_SetTextInsetFormatting(srObj, FV_EnclosingDoc);
        F_FilePathFree(textInsetPath);
    }
    break;
. . .
```

See also

- [“Sr_GetTextInsetFormatting\(\)” on page 177](#)
- [“Sr_GetTextInsetFilePath\(\)” on page 174](#)
- [“Sr_GetTextInsetFlowTag\(\)” on page 176](#)
- [“SrConvObjT” on page 415](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `IntT`

Sr_SetTextInsetPageSpace()

Specifies whether a FrameMaker text inset associated with an import conversion object of type `SR_OBJ_TEXT_INSET` is associated with body page or reference page space.

Synopsis

```
#include "fm_struct.h"

. . .
SrErrorT Sr_SetTextInsetPageSpace(SrConvObjT srObj,
    IntT pageSpace);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>pageSpace</i>	<code>FV_BodyPage</code> to associate the inset with a body page, or <code>FV_ReferencePage</code> to associate it with a reference page

Details

To specify that a FrameMaker text inset associated with an `SR_OBJ_TEXT_INSET` conversion object is associated with body page space or reference page space in the FrameMaker document, call `Sr_SetTextInsetPageSpace()`. *pageSpace* must be either `FV_BodyPage`, to assign the text inset to body page space, or `FV_ReferencePage`, to assign the text inset to reference page space. These enumerated types are defined in the `fapidefs.h` header file.

To determine the current page association for a text inset, call `Sr_GetTextInsetPageSpace()`.

Returns

On error, one of the following error codes:

- `SRW_E_BAD_VALUE`
- `SRW_E_WRONG_OBJ_TYPE`

Examples

The following code retrieves the page space for the text inset associated with the current conversion object. It ensures the page space is a reference page, and the flow is named "RefFlow."

```
. . .
FilePathT *textInsetPath;
IntT pageSpace;
. . .
```

```
case SR_EVT_BEGIN_ENTITY:
    if(textInsetPath = Sr_GetTextInsetFilePath(srObj)) {
        pageSpace = Sr_GetTextInsetPageSpace(srObj);
        if(pageSpace == FV_BodyPage) {
            Sr_SetTextInsetPageSpace(srObj, FV_ReferencePage);
            Sr_SetTextInsetFlowTag(srObj, (StringT)"RefFlow");
        }
        F_FilePathFree(textInsetPath);
    }
    break;
. . .
```

See also

- [“Sr_GetTextInsetPageSpace\(\)” on page 179](#)
- [“Sr_SetTextInsetFilePath\(\)” on page 228](#)
- [“Sr_SetTextInsetFlowTag\(\)” on page 231](#)
- [“SrConvObjT” on page 415](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `IntT`

Sr_SetVariableName()

Sets the name of a FrameMaker nonelement variable to associate with the specified conversion object of type `SR_OBJ_VARIABLE`.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_SetVariableName(SrConvObjT srObj, StringT varName);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>varName</i>	Variable name to assign

Details

This function sets the name of a FrameMaker nonelement variable associated with an import conversion object of type `SR_OBJ_VARIABLE`. The list of possible variable names is determined by the variables that are defined in the import template. For information about user defined variables and system variables in documents, see the *FrameMaker User's Guide*.

You can use read/write rules to specify that certain entities will be imported as FrameMaker variables. Usually, these rules are sufficient to convert the entity to a variable of a given name. In cases where the read/write rules are insufficient, you can use `Sr_SetVariableName()` to change the variable name for an import object.

For example, you might have an entity for DEPT_NAME. Assume FrameMaker creates a variable called DeptName, and defines it as the fully expressed department name. Whenever this entity appears in a table, you might want a department abbreviation. In that case, you can create a variable in the import template called DeptNameAbbr, and define it as the abbreviated department name. Then you can trap the DEPARTMENT_NAME entity, check to see if a table is currently open, and if so, use the DeptNameAbbr variable.

To change the proposed variable name, you must call this function before converting the object.

To retrieve the proposed variable name, call `Sr_GetVariableName()`.

Returns

On error, `SRW_E_WRONG_OBJ_TYPE`.

Examples

For an example of getting and setting variable names, see [“Sr_GetVariableName\(\)” on page 180](#).

See also

- [“Sr_GetVariableName\(\)” on page 180](#)
- [“SrConvObjT” on page 415](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `StringT`

Sr_UseFmElemId()

Specifies the FrameMaker element ID to use for the specified conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sr_UseFmElemId(SrConvObjT srObj, F_ObjHandleT elemId);
```

Arguments

<i>srObj</i>	Conversion object to change
<i>elemId</i>	ID of the FrameMaker element

Details

If your custom import client creates a FrameMaker element directly, using FDK calls, call `Sr_UseFmElemId()` to assign an ID for a container element or table object element. This function should be used only when you do not call `Sr_Convert()` to convert the import conversion object.

Returns

On error, one of the following error codes:

- `SRW_E_BAD_VALUE`
- `SRW_E_WRONG_OBJ_TYPE`

Examples

The following code traps markup elements named `author`. It then inserts a new FrameMaker element named “Contributor” as the first child of the currently open parent element, and sets that as the element to use. Finally, the code returns without converting the import event.

```
. . .
case SR_EVT_BEGIN_ELEM:
    if (F_StrIEqual(eventp->u.tag.gi, (StringT) "AUTHOR")) {
        SrInsertLocT insertLocation;
        SrConvObjT docObj;
        F_ObjHandleT firstChildId, docId, elemDefId, elemId;

        docObj = Sr_GetCurConvObjOfType(SR_OBJ_DOC);
        if (!docObj)
            docObj = Sr_GetCurConvObjOfType(SR_OBJ_BOOK_COMP);
        if (docObj)
            docId = Sr_GetDocId(docObj);
        else
            return(SRW_E_SUCCESS);
        elemDefId = F_ApiGetNamedObject(docId,
                                         FO_ElementDef, "Contributor");
    }
```

```
/* Be sure this element is the first child of parent */
insertLocation = Sr_GetInsertLoc(srObj);
firstChildId = F_ApiGetId(docId,
                        insertLocation.u.elemLoc.parentId,
                        FP_FirstChildElement);
insertLocation.u.elemLoc.childId = firstChildId;
insertLocation.u.elemLoc.offset = 0;
elemId = F_ApiNewElementInHierarchy(docId, elemDefId,
                                    &insertLocation.u.elemLoc);
Sr_UseFmElemId(srObj, elemId);
return(SRW_E_SUCCESS);
}
break;
```

. . .

See also

- [“Sr_GetFmElemId\(\)” on page 138](#)
- [“Sr_SetFmElemTag\(\)” on page 203](#)
- [“SrConvObjT” on page 415](#)
- [“SrwErrorT” on page 396](#)

Srw_CopyColSpec()

Copies a single *CALS* column specification (*colspec*).

Synopsis

```
#include "fm_struct.h"

. . .

SrwColSpecT Srw_CopyColSpec(SrwColSpecT *colspec);
```

Arguments

colspecp Pointer to the *colspec* to copy

Details

To make a copy of a single *colspec*, use *Srw_CopyColSpec()*. This function returns a structure containing duplicate *colspec* information.

When you no longer need the copy of the *colspec*, call *Srw_DeallocateColSpec()* to zero out the *colspec* structure and free the memory allocated it.

To copy a list of *colspecs*, use *Srw_CopyColSpecs()*.

Returns

An *SrwColSpecT* structure duplicating a *colspec*.

Examples

The following code copies a *colspec* and inserts it into a list of *colspecs*. Later it frees the memory allocated for them.

```
. . .
SrwColSpecT newColSpec, *originalColSpecP;
SrwColSpecsT colSpecList;
SrwErrorT errorStatus;

. . .
/* Copy a colspec. */
newColSpec = Srw_CopyColSpec(originalColSpecP);
/* Insert the copied colspec into a list. */
errorStatus = Srw_SetColSpec(&colSpecList, &newColSpec);

. . .
/* Cleanup--first free the colspec list. */
Srw_DeallocateColSpecs(&colSpecList);
/* Cleanup--now free the copied colspec. */
Srw_DeallocateColSpec(&newColSpec);

. . .
```

See also

- [“Srw_CopyColSpecs\(\)” on page 240](#)

- [“Srw_DeallocateColSpec\(\)” on page 250](#)
- [“SrwColSpecT” on page 418](#)

Srw_CopyColSpecs()

Copies a list of *CALS* column specifications (*colspecs*).

Synopsis

```
#include "fm_struct.h"
. . .
SrwColSpecsT Srw_CopyColSpecs(SrwColSpecsT *listp);
```

Arguments

listp Pointer to the list of *colspecs* to copy

Details

To make a copy of a list of *colspecs*, use `Srw_CopyColSpecs()`. This function returns a structure that contains a list of pointers to individual *colspec* structures.

When you no longer need the copy of the list, call `Srw_DeallocateColSpecs()` to zero out the list structure and free the memory allocated for it.

To copy an individual *colspec*, use `Srw_CopyColSpec()`.

Important: Do not deallocate individual *colspecs* pointed to within a *colspec* list. They are automatically freed when the list is deallocated.

Returns

An `SrwColSpecsT` structure duplicating a list of *colspecs*.

Examples

The following code copies a *colspec* list, and later frees the memory allocated for the copied list:

```
. . .
SrwColSpecsT newColSpecList, *originalColSpecListP;
. . .
/* Copy a colspec list. */
newColSpecList = Srw_CopyColSpecs(originalColSpecListP);
. . .
/* Cleanup--free the copied colspec list. */
Srw_DeallocateColSpecs(&newColSpecList);
. . .
```

See also

- [“Srw_CopyColSpec\(\)” on page 239](#)

- [“Srw_DeallocateColSpecs\(\)” on page 251](#)
- [“SrwColSpecsT” on page 419](#)

Srw_CopyPropVal()

Allocates memory for, and returns a copy of the passed `SrwPropValT` structure.

Synopsis

```
#include "fm_struct.h"
. . .
SrwPropValT Srw_CopyPropVal(SrwPropValT *propVal);
```

Arguments

propVal The property value to copy

Details

Conversion objects can have properties associated with them. The properties that can be assigned to an object depend on the conversion object type. The objects that can have properties are of the following types:

SR_OBJ_ELEM	SR_OBJ_TABLE	SR_OBJ_TABLE_TITLE
SR_OBJ_TABLE_HEADING	SR_OBJ_TABLE_BODY	SR_OBJ_TABLE_FOOTING
SR_OBJ_TABLE_ROW	SR_OBJ_TABLE_CELL	SR_OBJ_TABLE_COLSPEC
SR_OBJ_TABLE_SPANSPEC	SR_OBJ_GRAPHIC	SR_OBJ_EQUATION
SR_OBJ_VARIABLE	SR_OBJ_MARKER	SR_OBJ_XREF
SR_OBJ_FOOTNOTE		

For a list of the properties that apply the these object types, see [“SrwFmPropertyT” on page 397](#).

This function returns `SrwPropValT`, which is defined as:

```
typedef struct {
    SrwFmPropertyT prop;
    StringT value;
} SrwPropValT;
```

Note that the property value is a string, in spite of the fact that some of the properties are enumerated types or other numeric values. These numeric values are handled as strings to unify the format of all conversion object properties.

Important: Note that actual property values are strings. If you want to store a value in a variable, then you must either use `Sr_GetPropVal()`, or else use `F_StrCopyString()`, as follows:

```
propVal = F_StrCopyString(propVals.vals[i].value).
```

Important: When you are finished with the property value, be sure to call `Srw_DeallocatePropVal()` to free up a its memory.

Returns

A copy of the passed property value, or Null on error.

Examples

The following code traps an xref object, gets its list of properties, and makes a copy of the first property value in the list.

```
#include "fm_struct.h"
#include "fstrings.h"

SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    SrwPropValsT props;
    SrwPropValT copyProp;

    if (eventp->evtype == SR_EVT_BEGIN_ELEM) {
        if (Sr_GetObjType(srObj) == SR_OBJ_XREF) {
            props = Sr_GetPropVals(srObj);
            copyProp = Srw_CopyPropVal(&props.val[0]);
            Srw_DeallocatePropVal(&props);
            /* Use the copy as you wish. */

            . . .

            Srw_DeallocatePropVals(&copyProp);
        }
    }
    . . .
}
```

See also

- [“Sr_SetPropVals\(\)” on page 219](#)
- [“Sr_GetPropVal\(\)” on page 159](#)
- [“SrConvObjT” on page 415](#)
- [“SrwPropValsT” on page 421](#)
- [“SRW_errno” on page 418](#)

Srw_CopyPropVals()

Allocates memory for, and returns a copy of the passed `SrwPropValsT` structure.

Synopsis

```
#include "fm_struct.h"
. . .
SrwPropValsT Srw_CopyPropVals(SrwPropValsT *propVals);
```

Arguments

propVal The property value to copy

Details

Conversion objects can have properties associated with them. The properties that can be assigned to an object depend on the conversion object type. The objects that can have properties are of the following types:

SR_OBJ_ELEM	SR_OBJ_TABLE	SR_OBJ_TABLE_TITLE
SR_OBJ_TABLE_HEADING	SR_OBJ_TABLE_BODY	SR_OBJ_TABLE_FOOTING
SR_OBJ_TABLE_ROW	SR_OBJ_TABLE_CELL	SR_OBJ_TABLE_COLSPEC
SR_OBJ_TABLE_SPANSPEC	SR_OBJ_GRAPHIC	SR_OBJ_EQUATION
SR_OBJ_VARIABLE	SR_OBJ_MARKER	SR_OBJ_XREF
SR_OBJ_FOOTNOTE		

For a list of the properties that apply the these object types, see [“SrwFmPropertyT” on page 397](#).

This function returns `SrwPropValsT`, which is a list of all the properties for a specified conversion object. `SrwPropValsT` is defined as:

```
typedef struct {
    IntT len; /* Number of properties in array. */
    SrwPropValT *vals; /* Pointer to array of values. */
} SrwPropValsT;
```

Each property is expressed by a `SrwPropValT` structure, which is defined as:

```
typedef struct {
    SrwFmPropertyT prop;
    StringT value;
} SrwPropValT;
```

Note that the property value is a string, in spite of the fact that some of the properties are enumerated types or other numeric values. These numeric values are handled as strings to unify the format of all conversion object properties.

Important: Note that actual property values are strings. If you want to store a value in a variable, then you must either use `Sr_GetPropVal()`, or else use `F_StrCopyString()`, as follows:

```
propVal = F_StrCopyString(propVals.vals[i].value).
```

Important: When you are finished with the list of property values, be sure to call `Srw_DeallocatePropVals()` to free its memory. Use `Srw_DeallocatePropVal()` to free up a single property value.

Returns

A copy of the passed property values list, or a NULL on error.

Examples

The following code traps an xref object, gets its list of properties, and makes a copy of that list.

```
#include "fm_struct.h"
#include "fstrings.h"

SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    SrwPropValsT props, copyProps;

    if (eventp->evtype == SR_EVT_BEGIN_ELEM) {
        if (Sr_GetObjType(srObj) == SR_OBJ_XREF) {
            props = Sr_GetPropVals(srObj);
            copyProps = Srw_CopyPropVals(&props);
            Srw_DeallocatePropVals(&props);
            /* Use the copy as you wish. */
            . . .
            Srw_DeallocatePropVals(&copyProps);
        }
    }
    . . .
}
```

See also

- [“Sr_SetPropVals\(\)” on page 219](#)
- [“Sr_GetPropVal\(\)” on page 159](#)

- [“SrConvObjT” on page 415](#)
- [“SrwPropValsT” on page 421](#)
- [“SRW_errno” on page 418](#)

Srw_CopySpanSpec()

Copies a single *CALS* span specification (*spanspec*).

Synopsis

```
#include "fm_struct.h"
. . .
SrwSpanSpecT Srw_CopySpanSpec(SrwSpanSpecT *spanspec);
```

Arguments

spanspecp Pointer to the *spanspec* to copy

Details

To make a copy of a single *spanspec*, use `Srw_CopySpanSpec()`. This function returns a structure containing duplicate *spanspec* information.

When you no longer need the copy of the *spanspec*, call `Srw_DeallocateSpanSpec()` to zero out the *spanspec* structure and free the memory allocated it.

To copy a list of *spanspecs*, use `Srw_CopySpanSpecs()`.

Returns

An `SrwSpanSpecT` structure duplicating a *spanspec*.

Examples

The following code copies a *spanspec* and inserts it into a list of *spanspecs*. Later it zeroes out both the list and the copy of the *spanspec* and frees the memory allocated for them.

```
. . .
SrwSpanSpecT newSpanSpec, *originalSpanSpecP;
SrwSpanSpecsT spanSpecList;
SrwErrorT errorStatus;
. . .
/* Copy a spanspec. */
newSpanSpec = Srw_CopySpanSpec(originalSpanSpecP);
/* Insert the copied colspec into a list. */
errorStatus = Srw_SetSpanSpec(&spanSpecList, &newSpanSpec);
. . .
/* Cleanup--first free the spanspec list. */
Srw_DeallocateSpanSpecs(&spanSpecList);
```

```
/* Cleanup--now free the copied spanspec. */
Srw_DeallocateSpanSpec(&newSpanSpec);
. . .
```

See also

- [“Srw_CopySpanSpecs\(\)” on page 246](#)
- [“Srw_DeallocateSpanSpec\(\)” on page 254](#)
- [“SrwSpanSpecT” on page 422](#)

Srw_CopySpanSpecs()

Copies a list of CALS span specifications (*spanspecs*).

Synopsis

```
#include "fm_struct.h"
. . .
SrwSpanSpecsT Srw_CopySpanSpecs(SrwSpanSpecsT *listp);
```

Arguments

listp Pointer to the list of *spanspecs* to copy

Details

To make a copy of a list of *spanspecs*, use `Srw_CopySpanSpecs()`. This function returns a structure that contains a list of pointers to individual *spanspec* structures.

When you no longer need the copy of the list, call `Srw_DeallocateSpanSpecs()` to zero out the list structure and free the memory allocated for it.

To copy an individual *spanspec*, use `Srw_CopySpanSpec()`.

Important: Do not deallocate individual *spanspecs* pointed to within a *spanspec* list. They are automatically freed when the list is deallocated.

Returns

An `SrwSpanSpecsT` structure duplicating a list of *spanspecs*.

Examples

The following code copies a *spanspec* list, and later frees the memory allocated for the copied list:

```
. . .
SrwSpanSpecsT newSpanSpecList, *originalSpanSpecListP;
. . .
```

```
/* Copy a spanspec list */
newSpanSpecList = Srw_CopySpanSpecs(originalSpanSpecListP);
. . .
/* Cleanup--free the copied spanspec list */
Srw_DeallocateSpanSpecs(&newSpanSpecList);
. . .
```

See also

- [“Srw_CopySpanSpec\(\)” on page 245](#)
- [“Srw_DeallocateSpanSpecs\(\)” on page 255](#)
- [“SrwSpanSpecsT” on page 423](#)

Srw_CopyStraddle()

Copies a specified running straddle.

Synopsis

```
#include "fm_struct.h"
. . .
SrwStraddleT Srw_CopyStraddle(SrwStraddleT *stradp);
```

Arguments

stradp Pointer to the running straddle to copy

Details

To make a copy of a single running straddle, call `Srw_CopyStraddle()`. This function returns a structure containing duplicate running straddle information.

When you no longer need the copy of the straddle, call `Srw_DeallocateStraddle()` to zero out the straddle structure and free the memory allocated it.

To copy a list of straddles, use `Srw_CopyStraddles()`.

Returns

An `SrwStraddleT` structure duplicating a running straddle.

Examples

The following code copies a running straddle, and inserts it into a list of straddles. Later it zeroes out both the list and the copy of the running straddle and frees the memory allocated for them.

```
. . .
SrwStraddleT newStraddle, *originalStraddlep;
SrwStraddlesT straddleList;
SrwErrorT errorStatus;

. . .
/* Copy a running straddle. */
newStraddle = Srw_CopyStraddle(originalStraddlep);
/* Insert the copied running straddle into a list. */
errorStatus = Srw_SetStraddle(&straddleList, &newStraddle);

. . .
/* Cleanup--first free the running straddle list. */
Srw_DeallocateStraddles(&straddleList);
/* Cleanup--now free the copied straddle. */
Srw_DeallocateStraddle(&newStraddle);

. . .
```

See also

- [“Srw_CopyStraddles\(\)” on page 248](#)
- [“Srw_DeallocateStraddle\(\)” on page 256](#)
- [“SrwStraddleT” on page 423](#)

Srw_CopyStraddles()

Copies a list of running straddles.

Synopsis

```
#include "fm_struct.h"

. . .
SrwStraddlesT Srw_CopyStraddles(SrwStraddlesT *listp);
```

Arguments

listp Pointer to the running straddles list to copy

Details

To make a copy of a list of running straddles, call `Srw_CopyStraddles()`. This function returns a structure that contains a list of pointers to individual running-straddle structures.

When you no longer need the copy of the list, call `Srw_DeallocateStraddles()` to zero out the list structure and free the memory allocated for it.

To copy an individual running straddle, use `Srw_CopyStraddle()`.

Important: Do not deallocate individual running straddles referenced by a running straddle list. They are automatically freed when the list is deallocated.

Returns

An `SrwStraddlesT` structure duplicating a list of running straddles.

Examples

The following code copies a running straddle list, and later frees the memory allocated for the copied list:

```
. . .
SrwStraddlesT newStraddleList, *originalStraddleListp;
. . .
/* Copy a running straddle list. */
newStraddleList = Srw_CopyStraddles(originalStraddleListp);
. . .
/* Cleanup--free the copied running straddle list. */
Srw_DeallocateStraddles(&newStraddleList);
. . .
```

See also

- [“Srw_CopyStraddle\(\)” on page 247](#)
- [“Srw_DeallocateStraddles\(\)” on page 257](#)
- [“SrwStraddlesT” on page 424](#)

Srw_DeallocateColSpec()

Deletes a single *CALS* column specification (*colspec*) and frees the memory allocated for it.

Synopsis

```
#include "fm_struct.h"

. . .

VoidT Srw_DeallocateColSpec(SrwColSpecT *colspec);
```

Arguments

colspecp Pointer to the *colspec* to free

Details

To delete a single *colspec* structure, and free the memory allocated for it, call `Srw_DeallocateColSpec()`. Memory freed by this function is zeroed out.

Returns

VoidT.

Examples

The following code copies a *colspec*, and later zeroes out the *colspec* and frees the memory allocated for it:

```
. . .
SrwColSpecT newColSpec, *originalColSpec;
. . .
/* Copy a colspec. */
newColSpec = Srw_CopyColSpec(originalColSpec);
. . .
/* Free a colspec. */
Srw_DeallocateColSpec(&newColSpec);
. . .
```

See also

- [“Srw_DeallocateColSpecs\(\)” on page 251](#)
- [“SrwColSpecT” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `VoidT`

Srw_DeallocateColSpecs()

Deletes a list of *CALS* column specifications (*colspecs*) and frees the memory allocated for the list.

Synopsis

```
#include "fm_struct.h"

. . .

VoidT Srw_DeallocateColSpecs(SrwColSpecsT *listp);
```

Arguments

listp Pointer to the list of *colspecs* to free

Details

SrwColSpecsT is a list of *SrwColSpecT* structures. This function frees the memory allocated for each *colspec* in the list, and then frees the memory allocated for the list structure pointed to by *listp*. All memory allocated for strings and arrays is freed.

Returns

VoidT.

Examples

The following code copies a *colspec* and inserts it into a list of *colspecs*. Later it deletes both the list and the copy of the *colspec*, and then frees the memory allocated for the list and the *colspec*:

```
. . .
SrwColSpecT newColSpec, *originalColSpecp;
SrwColSpecsT colSpecList;
SrwErrorT errorStatus;

. . .
/* Copy a colspec. */
newColSpec = Srw_CopyColSpec(originalColSpecp);
/* Insert the copied colspec into a list. */
errorStatus = Srw_SetColSpec(&colSpecList, &newColSpec);

. . .
/* Cleanup--first free the colspec list. */
Srw_DeallocateColSpecs(&colSpecList);
/* Cleanup--now free the copied colspec. */
Srw_DeallocateColSpec(&newColSpec);

. . .
```

See also

- [“Srw_DeallocateColSpec\(\)” on page 250](#)
- [“SrwColSpecsT” on page 419](#)

- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `VoidT`

Srw_DeallocatePropVal()

Frees memory allocated for a specified property of a `FrameMaker` object.

Synopsis

```
#include "fm_struct.h"
. . .
Srw_DeallocatePropVal(SrwPropValT *propVal);
```

Arguments

propVal Pointer to the property value to free

Details

`Srw_DeallocatePropVal()` performs deep deallocation, freeing any strings that were allocated for this structure. This function should only be used to free memory allocated by your client to store property values. Do not try to free property values that were generated by `FrameMaker`.

To free memory for a list of property values, use `Srw_DeallocatePropVals()`.

Returns

`VoidT`.

Examples

The following code frees the memory allocated for an individual property value:

```
. . .
SrwPropValT propertyValue;
. . .
Srw_DeallocatePropVal(&propertyValue);
. . .
```

See also

- [“Srw_DeallocatePropVals\(\)” on page 253](#)
- [“SrwPropValT” on page 420](#)

Srw_DeallocatePropVals()

Frees memory allocated for the specified list of FrameMaker object properties.

Synopsis

```
#include "fm_struct.h"

. . .

VoidT Srw_DeallocatePropVals(SrwPropValsT *propVals);
```

Arguments

propVals Pointer to the property values list to free

Details

SrwPropValsT is a list of SrwPropValT structures. Srw_DeallocatePropVals() performs deep deallocation, completely freeing any SrwPropValT structures that were allocated for this list. This function should only be used to free memory allocated by your client to store property value lists. Do not try to free property values that were generated by FrameMaker.

To free memory allocated for a single property value, use Sr_DeallocatePropVal() instead.

Returns

VoidT.

Examples

The following code frees the memory allocated for a list of properties:

```
. . .
SrwPropValsT propertyList;
. . .
Srw_DeallocatePropVals(&propertyList);
. . .
```

See also

- [“Srw_DeallocatePropVal\(\)” on page 252](#)
- [“SrwPropValsT” on page 421](#)

Srw_DeallocateSpanSpec()

Deletes a single *CALS* span specification (*spanspec*) and frees the memory allocated for it.

Synopsis

```
#include "fm_struct.h"

. . .

VoidT Srw_DeallocateSpanSpec(SrwSpanSpecT *spanspecp);
```

Arguments

spanspecp Pointer to the *spanspec* to free

Details

Srw_DeallocateSpanSpec() performs deep deallocation, freeing any strings that were allocated for this structure.

Returns

VoidT.

Examples

The following code copies a *spanspec*. Later it deletes the *spanspec* and frees the memory allocated for it:

```
. . .
SrwSpanSpecT newSpanSpec, *originalSpanSpecp;
. . .
/* Copy a spanspec. */
newSpanSpec = Srw_CopySpanSpec(originalSpanSpecp);
. . .
/* Free the spanspec. */
Srw_DeallocateSpanSpec(&newSpanSpec);
. . .
```

See also

- [“Srw_DeallocateSpanSpecs\(\)” on page 255](#)
- [“SrwSpanSpecT” on page 422](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `VoidT`

Srw_DeallocateSpanSpecs()

Deletes a list of *CALS* span specifications (*spanspecs*) and frees the memory allocated for the list.

Synopsis

```
#include "fm_struct.h"

. . .

VoidT Srw_DeallocateSpanSpecs(SrwSpanSpecsT *listp);
```

Arguments

listp Pointer to the list of *spanspecs* to free

Details

SrwSpanSpecsT is a list of *SrwCSpanSpecT* structures. This function frees the memory allocated for each *spanspec* in the list, and then frees the memory allocated for the list structure pointed to by *listp*. All memory allocated for strings and arrays is freed.

Returns

VoidT.

Examples

The following code copies a *spanspec* and inserts it into a list of *spanspecs*. Later it deletes both the list and the copy of the *spanspec*, and frees the memory allocated for them.

```
. . .
SrwSpanSpecT newSpanSpec, *originalSpanSpecp;
SrwSpanSpecsT spanSpecList;
SrwErrorT errorStatus;

. . .
/* Copy a spanspec. */
newSpanSpec = Srw_CopySpanSpec(originalSpanSpecp);
/* Insert the copied spanspec into a list. */
errorStatus = Srw_SetSpanSpec(&spanSpecList, &newSpanSpec);

. . .
/* Cleanup--first free the spanspec list. */
Srw_DeallocateSpanSpecs(&spanSpecList);
/* Cleanup--now free the copied spanspec. */
Srw_DeallocateSpanSpec(&newSpanSpec);

. . .
```

See also

- [“Srw_DeallocateSpanSpec\(\)” on page 254](#)
- [“SrwSpanSpecsT” on page 423](#)

- [“Primitive data types” on page 383](#) for more information on `StringT` for more information on `VoidT`

Srw_DeallocateStraddle()

Deletes a single running straddle and frees the memory allocated for it.

Synopsis

```
#include "fm_struct.h"
. . .
VoidT Srw_DeallocateStraddle(SrwStraddleT *stradp);
```

Arguments

stradp Pointer to the running straddle to free

Details

This function performs a deep deallocation, freeing the structure, and the string that is allocated for the structure.

Returns

`VoidT`.

Examples

The following code copies a running straddle. Later it deletes the straddle and frees the memory allocated for it:

```
. . .
SrwStraddleT newStraddle, *originalStraddlep;
. . .
/* Copy a running straddle. */
newStraddle = Srw_CopyStraddle(originalStraddlep);
. . .
/* Free the running straddle. */
Srw_DeallocateStraddle(&newStraddle);
. . .
```

See also

- [“Srw_DeallocateStraddles\(\)” on page 257](#)
- [“SrwStraddleT” on page 423](#)
- [“Primitive data types” on page 383](#) for more information on `VoidT`

Srw_DeallocateStraddles()

Deletes a list of running straddles and frees the memory allocated for the list.

Synopsis

```
#include "fm_struct.h"

. . .

VoidT Srw_DeallocateStraddles(SrwStraddlesT *listp);
```

Arguments

listp Pointer to the list of running straddles to free

Details

SrwStraddlesT is a list of SrwStraddleT structures. This function performs a deep deallocation, freeing the SrwStraddlesT structure, and completely freeing the listed SrwStraddleT structures.

Returns

VoidT.

Examples

The following code copies a running straddle and inserts it into a list of straddles. Later it deletes the list and the copy of the running straddle, and frees the memory allocated for them:

```
. . .
SrwStraddleT newStraddle, *originalStraddlep;
SrwStraddlesT straddleList;
SrwErrorT errorStatus;

. . .
/* Copy a running straddle. */
newStraddle = Srw_CopyStraddle(originalStraddlep);
/* Insert the copied running straddle into a list. */
errorStatus = Srw_SetStraddle(&straddleList, &newStraddle);

. . .
/* Cleanup--first free the running straddle list. */
Srw_DeallocateStraddles(&straddleList);
/* Cleanup--now free the copied running straddle. */
Srw_DeallocateStraddle(&newStraddle);

. . .
```

See also

- [“Srw_DeallocateStraddle\(\)” on page 256](#)
- [“SrwStraddlesT” on page 424](#)

- [“Primitive data types” on page 383](#) for more information on `VoidT`

Srw_DeleteStraddlesByName()

Removes specified running straddles from a list of straddles.

Synopsis

```
#include "fm_struct.h"

. . .
VoidT Srw_DeleteStraddlesByName(SrwStraddlesT *listp,
StringT name);
```

Arguments

<i>listp</i>	Pointer to the list to search
<i>name</i>	Name of running straddles to remove

Details

To remove one or more instances of a specifically named running straddle from a list of straddles, call `Srw_DeleteStraddlesByName()`. This function removes the running straddle from the list of straddles, and frees the memory allocated for the running straddle's structure. Memory freed by this function is zeroed out.

Returns

`VoidT`.

Examples

The following code removes a named running straddle from a list of straddles:

```
. . .
SrwStraddlesT *straddleList;
. . .
Srw_DeleteStraddlesByName(straddleList, "Description");
. . .
```

See also

- [“Srw_DeallocateStraddle\(\)” on page 256](#)
- [“Srw_SetStraddle\(\)” on page 277](#)
- [“SrwStraddlesT” on page 424](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` and `VoidT`

Srw_EntityHandler()

Responds to external entity references. With this function, you can customize the handling for an external entity reference. For example, if the entity declaration specifies a *system ID* that is database key, you can use this handler to extract an appropriate file from the database to import into your FrameMaker document.

Synopsis

```
#include "fm_struct.h"

. . .
FilePathT *Srw_EntityHandler(
    StringT entName,
    StructuredEntityScopeT scope,
    FilePathT *defaultFp);
```

Arguments

<i>entName</i>	The name of the entity that is being referenced
<i>scope</i>	Whether the entity is a general or a parameter entity; one of <code>STRUCTURED_ES_GENERAL</code> or <code>STRUCTURED_ES_PARAMETER</code>
<i>defaultFp</i>	A pointer to the path for the file referenced by the entity

Details

`Srw_EntityHandler()` is a user-defined callback function that enables you to modify the way FrameMaker imports an entity, beyond the modifications permitted through FrameMaker read/write rules. For export, this handler returns the paths to any custom files you might have created so the software can validate the final markup instance.

On import, *entName*, *scope*, and *defaultFp* are determined by the entity declaration in the DTD. If the entity is declared in the markup document's internal DTD subset, FrameMaker stores the declaration data on the Entity Declarations reference page. After the declaration is stored, this handler can process the entity; as a result, FrameMaker saves the entity declaration as it was in the markup instance.

On export, the software uses the original entity declarations, generates the markup instance, and then validates it. If the declaration included a system identifier for the entity that does not resolve to a file, this final validation pass will generate errors. During the validation pass, this handler is called for each external entity reference. In that way, the validation can see the referenced file.

For example, assume the following entity declaration in the internal DTD subset of the markup instance:

```
<!ENTITY query1 SYSTEM "MyDatabaseQuery1">
```

Any reference to query (`&query1;`) does not resolve to a file. Assume the entity handler passes a query to a database that generates a markup fragment, and then it returns

the path to that generated text file. On import, FrameMaker saves the entity declaration on the Entity Declarations reference page. Then the handler generates the text file, returning the path to the text file. FrameMaker gets that filepath, and creates a text inset in the document.

On export, FrameMaker writes the entity declaration for `query` to the internal DTD subset of the markup instance. For every text inset associated with that entity, it writes out the entity reference, `&query;`. On the validation pass, the software uses the entity handler to locate the text file; in this way, it can fully validate the entity reference.

Note that your handler may need to get the full entity declaration. To do that, you pass *scope* and *entName* to `Structured_GetEntityDef()`.

If an entity refers to a FrameMaker document, this handler does not import the FrameMaker text. To accomplish that you must use the passed file path to get the FrameMaker file, then use FDK functions to import the content of a flow from that document into the document you are creating.

Returns

A pointer to a filepath, or `NULL` if there is no file path for the entity.

Examples

For more detailed examples, see [“Entity handlers” on page 25](#). The following code illustrates how an event handler might be set up to trap a database query, and return a path to a file generated by the database. Assume an entity declaration of `<!ENTITY query1 SYSTEM "MyDatabaseQuery1">`. If the handler finds a match for that entity definition, it processes the query and returns a path to the resulting file:

```
. . .
#include "fdetypes.h"
#include "futils.h"
#include "fm_struct.h"
. . .
FilePathT *Srw_EntityHandler(
    StringT entName,
    StructuredEntityScopeT scope,
    FilePathT *defaultFp)
{
    const StructuredEntityDefT *entDef;

    /* Use scope and entname to get the entity declaration. */
    entDef = Structured_GetEntityDef(scope, entName);
```

```
    if (F_StrIEqual(entDef->sysid, (StringT) "MyDatabaseQuery1"))
    {
        /* execute a database query that results in saving */
        /* an XML fragment to /temp_ents/query1 */
        . . .
        /* Then use the FDE function to convert a */
        /* device-independent path string to a */
        /* cross-platform FilePathT structure. */
        defaultFp = F_PathNameToFilePath(
            "<r><c>temp_ents<c>query1", NULL, FDIPath);
    }
    return defaultFp;
    /* No need to free entDef or defaultFp because they are */
    /* managed by FrameMaker. */
}
. . .
```

See also

- [“Structured GetEntityDef\(\)” on page 63](#)

Srw_GetColSpecByColNum()

Returns a *CALS* column specification (*colspec*) by number from a list of *colspecs*.

Synopsis

```
#include "fm_struct.h"
. . .
SrwColSpecT Srw_GetColSpecByColNum(SrwColSpecsT *listp,
    IntT colnum);
```

Arguments

<i>listp</i>	Pointer to the list to search
<i>colnum</i>	Column number of the <i>colspec</i> to return

Details

To retrieve a *colspec* structure based on its numeric position within a list of *colspecs*, call `Srw_GetColSpecByColNum()`.

CALS table column numbers start at 0, so to retrieve the first *colspec*, set *colnum* to 0. To retrieve the last *colspec* in a list, set *colnum* to one less than the number of items in the list.

After a retrieved *colspec* is no longer needed, free it with a call to `Srw_DeallocateColSpec()`.

You can also retrieve a *colspec* by name with `Srw_GetColSpecByName()`.

Returns

A populated `SrwColSpecT` structure describing the requested *colspec*, or a zeroed out data structure on error. On error, `SRW_errno` is set to `SRW_E_FAILURE`.

Examples

The following code retrieves the first *colspec* in a list, and later deallocates the *colspec* structure:

```
. . .
SrwColSpecT requestedColSpec;
SrwColSpecsT *colSpecList;

. . .
requestedColSpec = Srw_GetColSpecByColNum(colSpecList, 0);
. . .
/* Don't forget to free the colspec structure. */
Srw_DeallocateColSpec(&requestedColSpec);
. . .
```

See also

- [“Srw_GetColSpecByName\(\)” on page 262](#)
- [“Srw_DeallocateColSpec\(\)” on page 250](#)
- [“SrwColSpecT” on page 418](#)
- [“SrwColSpecsT” on page 419](#)
- [“SRW_errno” on page 418](#)

Srw_GetColSpecByName()

Returns a *CALS* column specification (*colspec*) by name from a list of *colspecs*.

Synopsis

```
#include "fm_struct.h"

. . .
SrwColSpecT Srw_GetColSpecByName(SrwColSpecsT *listp,
StringT name);
```

Arguments

<i>listp</i>	Pointer to the list to search
<i>name</i>	Name of the <i>colspec</i> to return

Details

To retrieve a *colspec* structure based on its column name within a list of *colspecs*, call `Srw_GetColSpecByName()`.

After a retrieved *colspec* is no longer needed, call `Srw_DeallocateColSpec()` to free it.

To retrieve a *colspec* by its numeric position within a list of *colspecs*, call `Srw_GetColSpecByColNum()`.

Returns

A populated `SrwColSpecT` structure describing the requested *colspec*, or a zeroed out data structure on error. On error, `SRW_errno` is set to `SRW_E_FAILURE`.

Examples

The following code retrieves a specified *colspec* in a list, and later deallocates the *colspec* structure:

```
. . .
SrwColSpecT requestedColSpec;
SrwColSpecsT *colSpecList;

. . .
requestedColSpec = Srw_GetColSpecByName(colSpecList,
    "Arguments");

. . .
/* Don't forget to free the colspec structure. */
Srw_DeallocateColSpec(&requestedColSpec);

. . .
```

See also

- [“Srw_GetColSpecByColNum\(\)” on page 261](#)
- [“Srw_DeallocateColSpec\(\)” on page 250](#)
- [“SrwColSpecT” on page 418](#)
- [“SrwColSpecsT” on page 419](#)
- [“SRW_errno” on page 418](#)

Srw_GetExportDtdFilePath()

Retrieves the platform-independent filepath of the DTD to use when exporting FrameMaker documents to markup. The export DTD is specified as part of the current structure application. All available structure applications for the current FrameMaker session are specified in the application definition file. For more information about the application file, see the *FrameMaker Structure Application Developer's Guide*.

Synopsis

```
#include "fm_struct.h"
#include "fdetypes.h"
#include "futils.h"

. . .
FilePathT *Srw_GetExportDtdFilePath(VoidT);
```

Arguments

None.

Details

FilePathT is a pointer to a platform-independent representation of a platform-specific path. You must include the FDE libraries to use this data type. For more information about working with platform-independent representations of filepaths, see Chapter 16, "Making I/O and Memory Calls Portable," in the *FDK Programmer's Guide*.

Returns

A pointer to the platform-independent filepath of the export DTD.

Examples

The following code retrieves the filepath for the export DTD that is specified for the current structure application:

```
. . .
FilePathT *exportPath;
. . .
exportPath = Srw_GetExportDtdFilePath();
. . .
F_FilePathFree(exportPath);
```

See also

- [“Srw_GetExportSchemaFilePath\(\)” on page 265](#)
- [“Srw_GetImportTemplateFilePath\(\)” on page 266](#)
- [“Srw_GetRulesDocFilePath\(\)” on page 268](#)
- [“Srw_GetStructuredDeclarationFilePath\(\)” on page 270](#)

- [“Srw_GetStructuredDocFilePath\(\)” on page 271](#)
- [“FilePathT” on page 409](#)

Srw_GetExportSchemaFilePath()

Retrieves the platform-independent filepath of the Schema to use when exporting FrameMaker documents to XML. The export Schema is specified as part of the current structure application. All available structure applications for the current FrameMaker session are specified in the application definition file. For more information about the application file, see the *FrameMaker Structure Application Developer's Guide*.

Synopsis

```
#include "fm_struct.h"
#include "fdetypes.h"
#include "futils.h"
. . .
FilePathT *Srw_GetExportSchemaFilePath(VoidT);
```

Arguments

None.

Details

FilePathT is a pointer to a platform-independent representation of a platform-specific path. You must include the FDE libraries to use this data type. For more information about working with platform-independent representations of filepaths, see Chapter 16, "Making I/O and Memory Calls Portable," in the *FDK Programmer's Guide*.

Returns

A pointer to the platform-independent filepath of the export Schema.

Examples

The following code retrieves the filepath for the export Schema that is specified for the current structure application:

```
. . .
FilePathT *exportPath;
. . .
exportPath = Srw_GetExportSchemaFilePath();
. . .
F_FilePathFree(exportPath);
```

See also

- [“Srw_GetExportDtdFilePath\(\)” on page 264](#)
- [“Srw_GetImportTemplateFilePath\(\)” on page 266](#)

- [“Srw_GetRulesDocFilePath\(\)” on page 268](#)
- [“Srw_GetStructuredDeclarationFilePath\(\)” on page 270](#)
- [“Srw_GetStructuredDocFilePath\(\)” on page 271](#)
- [“FilePathT” on page 409](#)

Srw_GetImportTemplateFilePath()

Retrieves the platform-independent filepath of the import template file. The import template is specified as part of the current structure application. All available structure applications for the current FrameMaker session are specified in the application definition file. For more information about the application file, see the *FrameMaker Structure Application Developer's Guide*.

Synopsis

```
#include "fm_struct.h"
#include "fdetypes.h"
#include "futils.h"

. . .

FilePathT *Srw_GetImportTemplateFilePath(VoidT);
```

Arguments

None.

Details

FilePathT is a pointer to a platform-independent representation of a platform-specific path. You must include the FDE libraries to use this data type. For more information about working with platform-independent representations of filepaths, see Chapter 16, "Making I/O and Memory Calls Portable," in the *FDK Programmer's Guide*.

Returns

A pointer to the platform-independent filepath of the current import template file.

Examples

The following code retrieves the path for the import template file that is specified for the current structure application:

```
. . .
FilePathT *templatePath;
. . .
if(eventp->evtype == SR_EVT_BEGIN_READER) {
    templatePath = Srw_GetImportTemplateFilePath();
. . .
}
```

```
. . .  
F_FilePathFree(templatePath);
```

See also

- [“Srw_GetExportDtdFilePath\(\)” on page 264](#)
- [“Srw_GetExportSchemaFilePath\(\)” on page 265](#)
- [“Srw_GetRulesDocFilePath\(\)” on page 268](#)
- [“Srw_GetStructuredDeclarationFilePath\(\)” on page 270](#)
- [“Srw_GetStructuredDocFilePath\(\)” on page 271](#)
- [“FilePathT” on page 409](#)

Srw_GetMainDocBookId()

Retrieves the object handle for the main FrameMaker document or book.

Synopsis

```
#include "fm_struct.h"  
  
. . .  
F_ObjHandleT Srw_GetMainDocBookId(VoidT);
```

Arguments

None.

Details

To retrieve the object handle for the main FrameMaker document or book, use `Srw_GetMainDocBookId()`. On import, the object is the document or book that imports its content from an markup document. On export, the object is the document from which markup content is derived.

If the FrameMaker object is a book, the ID returned by this function is the ID of the book file.

Returns

A document or book object handle.

Examples

The following case statement within an import event handler prints out the same ID's for the current book or document, and the main book or document. Note that for import, the corresponding book or document event must be converted before you can get an ID.

```
. . .  
SrConvObjT docObj, bookObj;  
F_ObjHandleT docId, bookId, mainDocBookId;  
. . .
```

```
case SR_EVT_BEGIN_DOC:
    bookObj = Sr_GetCurConvObjOfType(SR_OBJ_BOOK);
    if(bookObj)
        bookId = Sr_GetBookId(srObj);

    Sr_Convert(eventp, srObj);
    docObj = Sr_GetCurConvObjOfType(SR_OBJ_DOC);
    if (!docObj)
        docObj = Sr_GetCurConvObjOfType(SR_OBJ_BOOK_COMP);
    if(docObj)
        docId = Sr_GetDocId(docObj);
    mainDocBookId = Srw_GetMainDocBookId();
    if(bookObj) {
        F_Printf(NULL, "\nBOOK_ID: %d \nMAIN_DOC_BOOK_ID: %d",
            bookd, mainDocBookId);
    } else {
        F_Printf(NULL, "\nDOC_ID: %d \nMAIN_DOC_BOOK_ID: %d",
            docId, mainDocBookId);
    }
    return(SRW_E_SUCCESS);
    break;

. . .
```

See also

- [“Srw_GetStructuredDocFilePath\(\)” on page 271](#)
- [“Primitive data types” on page 383](#) for more information on F_ObjHandleT

Srw_GetRulesDocFilePath()

Retrieves the platform-independent filepath of the current read/write rules file. The rules file is specified as part of the current structure application. All available structure applications for the current FrameMaker session are specified in the application definition file. For more information about the application file, see the *FrameMaker Structure Application Developer's Guide*.

Synopsis

```
#include "fm_struct.h"
#include "fdetypes.h"
#include "futils.h"

. . .
FilePathT *Srw_GetRulesDocFilePath(VoidT);
```

Arguments

None.

Details

`FilePathT` is a pointer to a platform-independent representation of a platform-specific path. You must include the FDE libraries to use this data type. For more information about working with platform-independent representations of filepaths, see Chapter 16, "Making I/O and Memory Calls Portable," in the *FDK Programmer's Guide*.

The rules file governs either changes in default import behavior from markup to FrameMaker or changes in default behavior on export from FrameMaker to markup.

Returns

A pointer to the platform-independent filepath of the current read/write rules file.

Examples

The following code retrieves the filepath for the current read/write rules file.:

```
. . .
FilePathT *rulesPath;
. . .
rulesPath = Srw_GetRulesDocFilePath();
. . .
F_FilePathFree(rulesPath);
```

See also

- [“Srw_GetExportDtdFilePath\(\)” on page 264](#)
- [“Srw_GetExportSchemaFilePath\(\)” on page 265](#)
- [“Srw_GetImportTemplateFilePath\(\)” on page 266](#)
- [“Srw_GetStructuredDeclarationFilePath\(\)” on page 270](#)
- [“Srw_GetStructuredDocFilePath\(\)” on page 271](#)
- [“FilePathT” on page 409](#)

Srw_GetStructuredDeclarationFilePath()

Deprecated: *Srw_GetSgmlDeclarationFilePath()*

Retrieves the platform-independent filepath of the current SGML declaration.

Synopsis

```
#include "fm_struct.h"
#include "fdetypes.h"
#include "futils.h"
. . .
FilePathT *Srw_GetStructuredDeclarationFilePath(VoidT);
```

Arguments

None.

Details

FilePathT is a pointer to a platform-independent representation of a platform-specific path. You must include the FDE libraries to use this data type. For more information about working with platform-independent representations of filepaths, see Chapter 16, "Making I/O and Memory Calls Portable," in the *FDK Programmer's Guide*.

When you import a document, FrameMaker looks for the SGML declaration in the following order and locations:

1. An explicit declaration preceding the DTD in the document you import
2. A specification in a subset of your application
3. The default declaration provided as part of FrameMaker

When you export a document as SGML, you provide an SGML declaration as a subset of your application. The SGML declaration to use is specified as part of the current SGML application. All available structure applications for the current FrameMaker session are specified in the application definition file. For more information about the application file, see "Chapter 4, Working with Special Files" in the *FrameMaker Structure Application Developer's Guide*.

Returns

A pointer to the platform-independent filepath for the current SGML declaration. For XML returns NULL.

Examples

The following code retrieves the filepath for the current SGML declaration:

```
. . .
FilePathT *declaration;
. . .
declaration = Srw_GetStructuredDeclarationFilePath();
. . .
F_FilePathFree(declaration);
```

See also

- [“Srw_GetExportDtdFilePath\(\)” on page 264](#)
- [“Srw_GetExportSchemaFilePath\(\)” on page 265](#)
- [“Srw_GetImportTemplateFilePath\(\)” on page 266](#)
- [“Srw_GetRulesDocFilePath\(\)” on page 268](#)
- [“Srw_GetStructuredDocFilePath\(\)” on page 271](#)
- [“FilePathT” on page 409](#)

Srw_GetStructuredDocFilePath()

Deprecated: Srw_GetSgmlDocFilePath()

Retrieves the platform-independent filepath for the markup document being processed. When importing markup to a FrameMaker document, this returns the path to the markup document being opened or imported. When exporting a FrameMaker document to markup, this returns the filepath for the markup document being exported.

Synopsis

```
#include "fm_struct.h"
#include "fdetypes.h"
#include "futils.h"
. . .
FilePathT *Srw_GetStructuredDocFilePath(VoidT);
```

Arguments

None.

Details

FilePathT is a pointer to a platform-independent representation of a platform-specific path. You must include the FDE libraries to use this data type. For more information about working with platform-independent representations of filepaths, see Chapter 16, "Making I/O and Memory Calls Portable," in the *FDK Programmer's Guide*.

Returns

A pointer to the platform-independent filepath for an markup document.

Examples

The following code retrieves the filepath for the current markup document:

```
. . .
FilePathT *markupDoc;

. . .
markupDoc = Srw_GetStructuredDocFilePath();

. . .
F_FilePathFree(markupDoc);
```

See also

- [“Sr_GetBookFilePath\(\)” on page 117](#)
- [“Sr_SetBookFilePath\(\)” on page 190](#)
- [“Srw_GetExportDtdFilePath\(\)” on page 264](#)
- [“Srw_GetExportSchemaFilePath\(\)” on page 265](#)
- [“Srw_GetImportTemplateFilePath\(\)” on page 266](#)
- [“Srw_GetStructuredDeclarationFilePath\(\)” on page 270](#)
- [“Srw_GetMainDocBookId\(\)” on page 267](#)
- [“FilePathT” on page 409](#)

Srw_GetSpanSpecByName()

Returns a *CALS* span specification (*spanspec*) by name from a list of *spanspecs*.

Synopsis

```
#include "fm_struct.h"

. . .
SrwSpanSpecT Srw_GetSpanSpecByName(SrwSpanSpecsT *listp,
StringT name);
```

Arguments

<i>listp</i>	Pointer to the list to search
<i>name</i>	Name of the <i>spanspec</i> to return

Details

To retrieve a *spanspec* structure based on its span column name within a list of *spanspecs*, call `Srw_GetSpanSpecByName()`.

Srw_LogMessage()

After a retrieved *spanspec* is no longer needed, free it with a call to `Srw_DeallocateSpanSpec()`.

Returns

A populated `SrwSpanSpecT` structure describing the requested *spanspec*, or a zeroed out data structure on error. On error, `SRW_errno` is set to `SRW_E_FAILURE`.

Examples

The following code retrieves a specified *spanspec* in a list, and later deallocates the *spanspec* structure:

```
. . .
SrwSpanSpecT requestedSpanSpec;
SrwSpanSpecsT *spanSpecList;
. . .
requestedSpanSpec = Srw_GetSpanSpecByName(spanSpecList,
    "ElemObjTypes");
. . .
/* Don't forget to free the spanspec structure. */
Srw_DeallocateSpanSpec(&requestedSpanSpec);
. . .
```

See also

- [“Srw_DeallocateSpanSpec\(\)” on page 254](#)
- [“SrwSpanSpecT” on page 422](#)
- [“SrwSpanSpecsT” on page 423](#)
- [“SRW_errno” on page 418](#)

Srw_LogMessage()

Writes a message to the log file..

Synopsis

```
#include "fm_struct.h"
. . .
VoidT Srw_LogMessage(SrwLogMessageLocationT *location,
    StringT message);
```

Arguments

<i>location</i>	The document, and location within the document, to associate with the log message; NULL writes a message to the log file with no associated location
<i>message</i>	Message string to write to the log file

Details

A log file is actually a FrameMaker document. At the end of an import or export session, if anything was written to the log file FrameMaker displays the log as an unnamed, locked document. You can write a message to the log, as well as some indication of the location in the file that generated the log event.

If you are importing markup, the location associated with the message is expressed as a hypertext link from the log file to that location in the FrameMaker document. When exporting to markup, the location is expressed as the appropriate file path and line number in the exported markup document. You pass this information to the log file via `location`.

For more information about log files, see the *FrameMaker Structure Application Developer's Guide*.

Returns

VoidT.

Examples

The following code writes a message to the log file if the flag `ERROR_CONDITION` is set:

```
. . .
SrConvObjT srDocObj, srElemObj;
StringT logMessage;
SrwLogMessageLocationT msgLoc;
F_ObjHandleT docId, elemId;
IntT ERROR_CONDITION = 0;

logMessage = F_StrCopyString((StringT) "My message");
/* After trapping an error... */
if (ERROR_CONDITION) {
    srDocObj = Sr_GetCurConvObjOfType(SR_OBJ_DOC);
    docId = Sr_GetDocId(srDocObj);

    srElemObj = Sr_GetCurConvObjOfType(SR_OBJ_ELEM);
    elemId = Sr_GetFmElemId(srElemObj);

    /*Now set the values in msgLoc and log the message*/
    msgLoc.docIdentType = SRW_LOGD_ID;
    msgLoc.doc_u.docId = docId;
    msgLoc.locType = SRW_LOGL_NONE;

    Srw_LogMessage (&msgLoc, logMessage);
}
```

See also

- [“SrwLogMessageLocationT” on page 420](#)

- [“SrwErrorT” on page 396](#)
- [“SRW_errno” on page 418](#)

Srw_SetColSpec()

Inserts or updates a *CALS* column specification (*colspec*) in a list of *colspecs*.

Synopsis

```
#include "fm_struct.h"

. . .

SrwErrorT Srw_SetColSpec(SrwColSpecsT *listp,
    SrwColSpecT *colspecp);
```

Arguments

<i>listp</i>	Pointer to the list to search
<i>colspecp</i>	Pointer to the <i>colspec</i> to insert

Details

To insert a new *colspec* into a list of *colspecs*, or to update an existing *colspec* in a list with a new version, call `Srw_SetColSpec()`. If an entry for the specified *colspec* does not exist, this function appends the new *colspec* to the end of the list. If the entry for the specified *colspec* already exists, it is overwritten with the new *colspec*.

Returns

On error, `SRW_E_FAILURE`.

Examples

The following code inserts a new *colspec* into a list:

```
. . .
SrwErrorT errorStatus;
SrwColSpecsT existingList;
SrwColSpecT newColSpec;

. . .
errorStatus = Srw_SetColSpec(&existingList, &newColSpec);

. . .
/* Later, deallocate the colspec. */
Srw_DeallocateColSpec(&newColSpec);

. . .
```

See also

- [“Srw_SetSpanSpec\(\)” on page 276](#)
- [“Srw_SetStraddle\(\)” on page 277](#)

- [“SrwColSpecT” on page 418](#)
- [“SrwColSpecsT” on page 419](#)
- [“SrwErrorT” on page 396](#)

Srw_SetSpanSpec()

Inserts or updates a *CALS* span specification (*spanspec*) in a list of *spanspecs*.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Srw_SetSpanSpec(SrwSpanSpecsT *listp,
    SrwSpanSpecT *spanspecp);
```

Arguments

<i>listp</i>	Pointer to the list to search
<i>spanspecp</i>	Pointer to the <i>spanspec</i> to insert

Details

To insert a new *spanspec* into a list of *spanspecs*, or to update an existing *spanspec* in a list with a new version, call `Srw_SetSpanSpec()`. If an entry for the specified *spanspec* does not exist, this function appends the new *spanspec* to the end of the list. If the entry for the specified *spanspec* already exists, it is overwritten with the new *spanspec*.

Returns

On error, `SRW_E_FAILURE`.

Examples

The following code inserts a new *spanspec* into a list:

```
. . .
SrwErrorT errorStatus;
SrwSpanSpecsT existingList;
SrwSpanSpecT newSpanSpec;
. . .
errorStatus = Srw_SetSpanSpec(&existingList, &newSpanSpec);
. . .
/* Later, deallocate the spanspec. */
Srw_DeallocateSpanSpec(&newSpanSpec);
. . .
```

See also

- [“Srw_SetColSpec\(\)” on page 275](#)

- [“Srw_SetStraddle\(\)” on page 277](#)
- [“SrwSpanSpecT” on page 422](#)
- [“SrwSpanSpecsT” on page 423](#)
- [“SrwErrorT” on page 396](#)

Srw_SetStraddle()

Inserts or updates a running straddle in a list of straddles.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Srw_SetStraddle(SrwStraddlesT *listp,
SrwStraddleT *stradp);
```

Arguments

<i>listp</i>	Pointer to the list to search
<i>stradp</i>	Pointer to the running straddle to insert

Details

To insert a new running straddle into a list of straddles, or to update an existing straddle in a list with a new version, call `Srw_SetStraddle()`. If an entry for the specified running straddle does not exist, this function appends the new straddle to the end of the list. If the entry for the specified straddle already exists, it is overwritten with the new running straddle.

Returns

On error, `SRW_E_FAILURE`.

Examples

The following code inserts a new running straddle into a list:

```
. . .
SrwErrorT errorStatus;
SrwStraddlesT ExistingList;
SrwStraddleT NewStraddle;`
. . .
errorStatus = Srw_SetStraddle(&ExistingList, &NewStraddle);
. . .
/* Later, deallocate the straddle */
Srw_DeallocateStraddle(&NewStraddle);
```

See also

- [“Srw_DeleteStraddlesByName\(\)” on page 258](#)

- [“Srw_SetColSpec\(\)” on page 275](#)
- [“Srw_SetSpanSpec\(\)” on page 276](#)
- [“SrwStraddleT” on page 423](#)
- [“SrwStraddlesT” on page 424](#)
- [“SrwErrorT” on page 396](#)

StartTagOmissible()

Tests whether an element’s start-tag can be omitted in a document instance.

Synopsis

```
#include "fm_struct.h"
. . .
IntT StartTagOmissible (StructuredElementDefT *elemDef);
```

Arguments

elemDef Pointer to the element definition to query

Details

`StartTagOmissible()` is a macro that tests the `tagOmission` field of *elemDef* to determine if `OMIT_START_TAG` is set. If set, it indicates that the start-tag can be omitted.

Returns

1 if the start-tag is omissible; otherwise, 0. For XML returns 0.

Examples

The following code executes a statement only if the start-tag can be omitted for the `memo` element:

```
. . .
const StructuredElementDefT *elemDefp;
ConStringT elemName;
. . .
switch(eventp->evtype) {
case SW_EVT_BEGIN_DOC:
. . .
    elemDefp = Structured_GetElementDef((StringT)"memo");
    if (StartTagOmissible (elemDefp)) {
        /* tag can be omitted, so... */
        . . .
    }
. . .
```

See also

- [“EndTagOmissible\(\)” on page 50](#)
- [“Primitive data types” on page 383](#) for more information on `BoolT`

StartTagOmissible()

Sw_CancelCurBatchFile()

When exporting markup files in batch mode, terminates export of the current FrameMaker document and continues processing the next document.

Synopsis

```
#include "fm_struct.h"

. . .

VoidT Sw_CancelCurBatchFile(VoidT);
```

Arguments

None.

Details

This function stops exporting the current FrameMaker document at the end of the current event, and allows batch processing to resume with the next file. When exporting a single document file, terminates the export and returns control to FrameMaker.

A batch export process is one that operates on an entire directory of FrameMaker files. For example, users can start a batch process by choosing Convert Documents to Structured Format from the File > Utilities menu. Unix users can also start a batch process from the command line.

Use this function when you encounter an error condition, for example, if you cannot write a graphic that was imported by reference to the proposed file path.

To cancel export of all FrameMaker documents during batch processing, call `Sw_CancelOperation()`.

Returns

VoidT.

Examples

The following code cancels processing of the current FrameMaker document if you cannot write to the proposed filepath for a graphic that was imported into the FrameMaker document by reference:

```
. . .
FilePathT *entFilePath;
StringT entityName;
Sw_ConvObjT docGraphic;
IntT flag = FF_FilePathWritable;
. . .
entFilePath = Sw_GetExportFilePath(docGraphic);
```

```
/* Test that the path is writable */
if (F_FilePathProperty(entFilePath, flag) != flag) {
    F_FilePathFree(entFilePath);
    Sw_CancelCurBatchFile();
    return (SRW_E_FAILURE);
} else
    F_FilePathFree(entFilePath);
. . .
```

See also

- [“Sw_CancelOperation\(\)” on page 282](#)
- [“Sw_GetSessionProps\(\)” on page 329](#) for information on getting the current batch mode
- [“Primitive data types” on page 383](#) for more information on VoidT

Sw_CancelOperation()

When exporting in batch mode, terminates the export of all FrameMaker documents and returns control to FrameMaker. When exporting a single document file, terminates the export and returns control to FrameMaker.

Synopsis

```
#include "fm_struct.h"
. . .
VoidT Sw_CancelOperation(VoidT);
```

Arguments

None.

Details

This function stops exporting the current FrameMaker document at the end of the current event, and returns control to FrameMaker. You can call this function when exporting a single document file, or when exporting in batch mode. If the current session is in batch mode, a number of files may have already been exported; this function does not affect any documents that were already exported by the batch process.

Use this function when you encounter an error condition, for example, if you cannot write a graphic that was imported by reference to the proposed file path.

To terminate export of a single FrameMaker document during a batch operation, call `Sw_CancelCurBatchFile()`.

Returns

VoidT.

Examples

The following code cancels processing of the current FrameMaker document if you cannot write to the proposed filepath for a graphic that was imported into the FrameMaker document by reference:

```
. . .
FilePathT *entFilePath;
StringT entityName;
Sw_ConvObjT docGraphic;
IntT flag = FF_FilePathWritable;

. . .
entFilePath = Sw_GetExportFilePath(docGraphic);
/* Test that the path is writable */
if (F_FilePathProperty(entFilePath, flag) != flag) {
    F_FilePathFree(entFilePath);
    Sw_CancelOperation();
    return (SRW_E_FAILURE);
} else
    F_FilePathFree(entFilePath);
. . .
```

See also

- [“Sw_CancelCurBatchFile\(\)” on page 281](#)
- [“Sw_GetSessionProps\(\)” on page 329](#) for information on getting the current batch mode
- [“Primitive data types” on page 383](#) for more information on VoidT

Sw_Convert()

Executes the specified conversion for the specified export conversion object.

Synopsis

```
#include "fm_struct.h"

. . .
SrwErrorT Sw_Convert(SwEventT *eventp, SwConvObjT swObj);
```

Arguments

<i>eventp</i>	Pointer to conversion event to perform
<i>swObj</i>	Conversion object to convert

Details

Call `Sw_Convert()` to convert modified or unmodified conversion objects from within a custom event handler. *eventp* specifies the conversion event to carry out. *swObj* is the conversion object upon which the conversion is to be performed.

Returns

On error, `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code converts the current event/conversion object pair:

```
. . .
SrweErrorT errorStatus;
. . .
errorStatus = Sw_Convert(eventp, swObj);
. . .
```

See also

- [“Sw_EventHandler\(\)” on page 286](#)
- [“SwConvObjT” on page 424](#)
- [“SwEventT” on page 425](#)
- [“SrweErrorT” on page 396](#)

Sw_DeallocateSessionProps()

Deletes an export session properties structure and frees the memory allocated for it. This function performs a deep deallocation, freeing any arrays or strings referenced by the `SwSessionPropsT` structure.

Synopsis

```
#include "fm_struct.h"
. . .
VoidT Sw_DeallocateSessionProps(SwSessionPropsT *sessionProps);
```

Arguments

sessionProps Pointer to session properties structure to free

Returns

`VoidT`.

Examples

The following code gets the `Id` for the current session conversion object, then retrieves the session properties for export FrameMaker documents. Finally, the code frees the space allocated to the `SwSessionPropsT` structure:

```
. . .
SwSessionPropsT exportSessionProps;
SwConvObjT swObj;
```

```
. . .
swObj = SwGetCurConvObjOfType(SW_OBJ_SESSION);
exportSessionProps = Sw_GetSessionProps(swObj);
. . .
Sw_DeallocateSessionProps(&exportSessionProps);
. . .
```

See also

- [“Sw_GetSessionProps\(\)” on page 329](#)
- [“Sw_SetSessionProps\(\)” on page 370](#)
- [“SwSessionPropsT” on page 425](#)
- [“Primitive data types” on page 383](#) for more information on VoidT

Sw_DeallocateTextItems()

Deletes a text items structure.

Synopsis

```
#include "fm_struct.h"
. . .
VoidT Sw_DeallocateTextItems(SwTextItemsT *textItems);
```

Arguments

textItems Pointer to text items structure to free

Details

SwTextItemsT is a list of SwTextItemT structures. This function performs a deep deallocation, freeing the SwTextItemsT structure, and completely freeing all the SwTextItemT structures that were allocated for it.

Returns

VoidT.

Examples

The following code deallocates a text items structure. For more detailed examples of handling SwTextItemsT structures, see [“Sw_GetStructuredText\(\)” on page 332](#) and [“Sw_SetStructuredText\(\)” on page 373](#).

```
. . .
SwTextItemsT textForExport;
. . .
Sw_DeallocateTextItems(&textForExport);
. . .
```

See also

- [“Sw_GetStructuredText\(\)” on page 332](#)
- [“Sw_SetStructuredText\(\)” on page 373](#)
- [“SwTextItemsT” on page 426](#)
- [“Primitive data types” on page 383](#) for more information on `VoidT`

Sw_EventHandler()

Responds to event/conversion pairs passed from FrameMaker to a custom export client. `Sw_EventHandler()` is the user-defined entry point for all structure import/export API-client export conversion applications.

Synopsis

```
#include "fm_struct.h"

. . .

SrwErrorT Sw_EventHandler(SwEventT *eventp, SwConvObjT swObj);
```

Arguments

<i>eventp</i>	Pointer to conversion event to perform
<i>swObj</i>	Conversion object to convert

Details

`Sw_EventHandler()` is a user-defined callback function that enables you to modify the behavior of FrameMaker document export behavior beyond the modifications permitted through FrameMaker read/write rules. All structure import/export API clients that modify export behavior must define `Sw_EventHandler()`. During export processing, FrameMaker calls this function to permit it to alter actions it proposes to undertake.

Your `Sw_EventHandler()` function should include code for all export modifications you want to make.

Returns

On error, `SRW_E_FAILURE`.

Examples

The following code illustrates how an event handler might be set up to modify column-width specifications in a *CALS* table:

```
. . .
SrwErrorT Sw_EventHandler(SwEventT *eventp, SwConvObjT swObj)
{
    SrwColSpecsT colspecs;
    SrwColSpecT colspec;
    SwConvObjT tblObj;
    switch(Sw_GetObjType(swObj))
    {
        case SW_OBJ_TABLE_ROW:

            if(eventp->evtype == SW_EVT_END_TABLE)
            {
                /* Retrieve current table. */
                tblObj = Sw_GetCurConvObjOfType(SW_OBJ_TABLE);
                /* Retrieve copy of colspecs for table object. */
                colspecs = Sw_GetColSpecs(tblObj);
                /* If there are no colspecs, exit gracefully. */
                if(colspecs.len == 0)
                    break;
                /* Retrieve copy of colspec for first column. */
                colspec = Srw_GetColSpecByColNum(&colspecs, 0);
                if(SRW_errno == SRW_E_SUCCESS)
                {
                    /* Set the column width to 12 cm. */
                    F_Free(colspec.width);
                    colspec.width = F_StrCopyString("12cm");
                    colspec.valueSet |= SRW_COLSPEC_COLWIDTH;
                    /* Put new colspec back into list. */
                    Srw_SetColSpec(&colspecs, &colspec);
                    /* Put the new list back into the object. */
                    Sw_SetColSpecs(tblObj, &colspecs);
                }
                /* Free any allocated memory. */
                Srw_DeallocateColSpec(&colspec);
                Srw_DeallocateColSpecs(&colspecs);
            }
            break;
        . . .
    }
}
```

```
        /* Convert object and return control to FrameMaker. */
        return Sw_Convert(eventp, swObj);
    }
```

See also

- [“Sw_Convert\(\)” on page 283](#)
- [“SwConvObjT” on page 424](#)
- [“SwEventT” on page 425](#)
- [“SwErrorT” on page 396](#)

Sw_GetAssociatedEvent()

Returns an export event associated with a specified export conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
SwEventT *Sw_GetAssociatedEvent(SwConvObjT swObj);
```

Arguments

swObj Conversion object for which to return an event

Details

When an export conversion object is created, it and its related event are pushed onto the conversion object stack. `Sw_GetAssociatedEvent()` retrieves a pointer to the event associated with a specific export conversion object. The event structure describes the data in the FrameMaker document that triggered the event.

This function is often used in conjunction with `Sw_GetCurConvObjOfType()` to retrieve information about a previously processed event/conversion object pair. For example, you might want to know the element tag for the parent of the current element. To do that, you get the parent conversion object, and then get the markup tag from its associated event.

Returns

The `SwEventT` data structure, or a NULL pointer on error. The `SwEventT` structure is defined as:

```
typedef struct {
    SwEventTypeT evtype;
    F_TextLocT txtloc;
    F_ObjHandleT fm_elemid;
    F_AttributesT fm_attrs;
    F_ObjHandleT fm_objid;
    StringT text;
    UCharT charcode;
    StringT chartag;
} SwEventT;
```

For a complete description of `SwEventT`, see [“SwEventT” on page 425](#).

On error, `SRW_errno` is set to one of the following values:

- `SRW_E_INVALID_CONV_OBJ`
- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_BAD_OBJ_HANDLE`

Examples

The following code finds the conversion object for the highest-level element in the currently open FrameMaker document. It then gets the event associated with that object.

```
. . .
SrwErrorT Sw_EventHandler(eventp, swObj)
SwEventT *eventp;
SwConvObjT swObj;
{
    SwConvObjT elemObj, docObj;
    SwEventT *elemEvtp;

    if (docObj = Sw_GetCurConvObjOfType(SW_OBJ_DOC)) {
        elemObj = Sw_GetChildConvObj(docObj);
        while (Sw_GetObjType(elemObj) != SW_OBJ_ELEM) {
            elemObj = Sw_GetChildConvObj(elemObj);
        }
        elemEvtp = Sw_GetAssociatedEvent(elemObj);
        rootId = elemEvtp->fm_objid;
    }
. . .
```

See also

- [“Sw_GetCurConvObjOfType\(\)” on page 303](#)

- [“SwConvObjT” on page 424](#)

Sw_GetAttrVal()

Retrieves the attribute-value data structure for a specified attribute of an export conversion object.

Synopsis

```
#include "fm_struct.h"

. . .
StructuredAttrValT Sw_GetAttrVal(SwConvObjT swObj,
StringT sgmlAttrName);
```

Arguments

<i>swObj</i>	Conversion object to query
<i>sgmlAttrName</i>	Name of the attribute to query

Details

This function retrieves the proposed value for a specified attribute. The value must have been assigned to the FrameMaker element that is being exported.

By default, a FrameMaker attribute translates to an attribute of the same name in markup. However, the current read/write rules can map the exported attribute to an attribute with a different name in markup. The attribute name you provide must match the name of the FrameMaker attribute. For more information about how FrameMaker translates attributes, see *FrameMaker Structure Application Developer's Guide*.

This function returns a structure of type `StructuredAttrValT`, which is defined as:

```
typedef struct (
    StringT sgmlAttrName;
    StringListT sgmlAttrVal;
    IntT sgmlAttrFlags;
} StructuredAttrValT;
```

Note that `sgmlAttrVal` is a string list. Attribute values in markup can be character data, or they can be tokens or lists of tokens. If the value is character data or a single token, then there is only one element in the string list (`F_StrListLen(sgmlAttrVal) == 1`). If the value is a list of tokens, then each string in the list corresponds to a single value token.

If you call `Sw_GetAttrVal()` before `swObj` is converted into an element in markup, the function returns the attribute value currently proposed for the element. If you call

`Sw_GetAttrVal()` after `swObj` is converted, the function returns the attribute value that was exported.

Important: When you no longer need the attribute-value data structure returned by this function, call `Structured_DeallocateAttrVal()` to delete the structure and free the memory allocated for it.

To get a list of attribute-values associated with a conversion object, call `Sw_GetAttrVals()`.

Returns

The requested attribute-value data structure, or a zeroed-out data structure on error. On error, `SRW_errno` is set to one the following error codes:

- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_NO_SUCH_ATTR`

Examples

The following code retrieves the attribute-value data structure for an `ACCESS` attribute associated with the current conversion object. It then checks the list of value tokens and if both the `TOP_SECRET` and the `INTERNAL` token are present it drops that element.

```
. . .
SrwErrorT Sw_EventHandler(eventp, swObj)
SwEventT *eventp;
SwConvObjT swObj;
{
    StructuredAttrValT curAttr;
    IntT i, flags, errorStatus;
    BoolT topSecret, secret, internal, general;

    topSecret = secret = internal = general = False;
```

```
if (eventp->evtype == SW_EVT_BEGIN_ELEM) {
    curAttr = Sw_GetAttrVal(swObj, (StringT) "ACCESS");
    for (i=0; i<F_StrListLen(curAttr.sgmlAttrVal); i++) {
        ConStringT curVal = F_StrListGet(curAttr.sgmlAttrVal, i);
        if (F_StrIEqual((StringT)"TOP_SECRET", curVal))
            topSecret = True;
        if (F_StrIEqual((StringT)"SECRET", curVal))
            secret = True;
        if (F_StrIEqual((StringT)"INTERNAL", curVal))
            internal = True;
        if (F_StrIEqual((StringT)"GENERAL", curVal))
            general = True;
    }
    /* When it is no longer needed, free the attribute. */
    Structured_DeallocateAttrVal(&curAttr);
    if (topSecret && internal) {
        /*Drop this element completely */
        flags = (SRW_DROP_ELEMENT_CONTENT | SRW_UNWRAP_ELEMENT);
        errorStatus = Sw_SetProcessingFlags(swObj, flags);
    }
    return (Sw_Convert(eventp, swObj));
}
```

See also

- [“Structured_DeallocateAttrVal\(\)” on page 55](#)
- [“Sw_SetAttrVal\(\)” on page 345](#)
- [“Sw_GetAttrVals\(\)” on page 293](#)
- [“SwConvObjT” on page 424](#)
- [“StructuredAttrValT” on page 412](#)
- [“Primitive data types” on page 383](#) for more information on StringT

Sw_GetAttrVals()

Retrieves the list of attribute values for the specified conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

StructuredAttrValsT Sr_GetAttrVals(SrConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

This function retrieves the list of proposed values for a specified conversion object. The object can only have attribute values if:

- The specified conversion object is associated with an event of type `SR_EVT_BEGIN_ELEM`.
- The corresponding markup element specifies at least one attribute, and includes a value for the attribute with the element's start tag.
- The import template's EDD includes appropriate attribute definitions for the proposed FrameMaker element.

By default, an attribute in markup translates to a FrameMaker attribute of the same name. However, the current read/write rules can map the imported attribute to a FrameMaker attribute with a different name. The attribute name you provide must match the name of the FrameMaker attribute. For more information about how FrameMaker translates attributes, see *FrameMaker Structure Application Developer's Guide*.

This function returns a structure of type `StructuredAttrValsT`, which is defined as:

```
typedef struct {
    UIntT len; /* number of attributes pairs */
    StructuredAttrValT *val; /* attribute pair array pointer */
} StructuredAttrValsT;
```

val points to an array of attribute values, each one represented by a structure of type `StructuredAttrValT`, which is defined as:

```
typedef struct (
    StringT sgmlAttrName;
    StringListT sgmlAttrVal;
    IntT sgmlAttrFlags;
} StructuredAttrValT;
```

For more information about single attribute values, see [“Sw_GetAttrVal\(\)” on page 290](#).

If you call `Sr_GetAttrVals()` before `swObj` is converted into a `FrameMaker` element, the function returns the proposed list of attribute values. If you call `Sr_GetAttrVals()` after `swObj` is converted, the function returns the list that was imported.

Important: When you no longer need the attribute-value data structure returned by this function, call `Structured_DeallocateAttrVals()` to delete the structure and free the memory allocated for it.

Returns

A list of attribute-value pairs, or a zeroed-out data structure on error. Not that a null structure can also indicate that the current object indicates an element that has no attributes defined for it. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code retrieves the list of attribute values associated with the current conversion object. It then loops through each list of attribute values and prints out the name of each attribute. Finally, it deallocates the attribute values structure.

```
. . .
SrwErrorT Sr_EventHandler(eventp, swObj)
SrEventT *eventp;
SrConvObjT swObj;
{
    StructuredAttrValsT curAttrVals;
    UIntT i;

    if (eventp->evtype == SR_EVT_BEGIN_ELEM) {
        curAttrVals = Sw_GetAttrVals(swObj);
        for (i=0; i < curAttrVals.len; i++) {
            F_Printf(NULL, "\nAttr Val Name: %s",
                    curAttrVals.val[i].sgmlAttrName);
        }
        Structured_DeallocateAttrVals(&curAttrVals);
    }
    return (Sw_Convert(eventp, swObj));
}
```

See also

- [“Sw_SetAttrVals\(\)” on page 347](#)
- [“Sw_GetAttrVal\(\)” on page 290](#)
- [“Structured_DeallocateAttrVals\(\)” on page 56](#)
- [“F_AttributesT” on page 410](#)
- [“SrConvObjT” on page 415](#)

- [“SRW_errno” on page 418](#)

Sw_GetBookCompEntityFilePath()

Retrieves the entity filepath for exporting a book component for the specified export conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
FilePathT *Sw_GetBookCompEntityFilePath(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

To retrieve the entity filepath for exporting a FrameMaker book component (a document that is part of a book), call `Sw_GetBookCompEntityFilePath()`. *swObj* must be of type `SW_OBJ_BOOK_COMP`.

`FilePathT` is a pointer to a platform-independent representation of a platform-specific path. For more information about working with platform-independent representations of filepaths, see Chapter 16, "Making I/O and Memory Calls Portable," in the *FDK Programmer's Guide*.

Returns

A pointer to the platform-independent filepath, or `NULL` on error. On error, `SRW_errno` is set to one of the following error codes:

- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_NOT_BOOK_COMP`

Examples

The following code retrieves the filepath for a book component associated with the current conversion object if it is of type `SW_OBJ_BOOK_COMP`:

```
. . .
FilePathT *bookCompFp;
. . .
if(Sw_GetObjType(swObj) == SW_OBJ_BOOK_COMP)
    bookCompFp = Sw_GetBookCompEntityFilePath(swObj);
. . .
```

See also

- [“Sw_SetBookCompEntityFilePath\(\)” on page 350](#)

- [“SwConvObjT” on page 424](#)
- [“FilePathT” on page 409](#)
- [“SRW_errno” on page 418](#)

Sw_GetBookCompPi()

Gets the markup document processing instructions (*PI*) for a FrameMaker book component associated with an export conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
StringT Sw_GetBookCompPi(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

To retrieve the proposed *PI* for a FrameMaker book component (a document that is part of a FrameMaker book), call `Sw_GetBookCompPi()`. *swObj* should be of type `SW_OBJ_BOOK_COMP`. The data returned is the *PI* without open (*PIO*) and close (*PIC*) delimiters.

To retrieve processing instructions for an entire FrameMaker book, rather than for a book component, call `Sw_GetBookPi()`.

Returns

The *PI* for the FrameMaker book component, or `NULL` on error. On error, `SRW_errno` is set to one of the following error codes:

- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_NOT_BOOK_COMP`

Examples

The following code retrieves the *PI* for the current conversion object if it is of type `SW_OBJ_BOOK_COMP`:

```
. . .
if (Sw_GetObjType(swObj) == SW_OBJ_BOOK_COMP) {
    StringT piStr = Sw_GetBookCompPi(swObj);
    . . .
    F_ApiDeallocateString(&piStr);
}
. . .
```


See also

- [“Sw_SetBookCompPi\(\)” on page 351](#)
- [“Sw_GetBookPi\(\)” on page 297](#)
- [“SwConvObjT” on page 424](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_GetBookPi()

Retrieves the markup *processing instructions (PI)* for a FrameMaker book file associated with an export conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
StringT Sw_GetBookPi(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

To retrieve the *PI* for a FrameMaker book file, call `Sw_GetBookPi()`. *swObj* should be of type `SW_OBJ_BOOK`. The data returned is the *processing instructions* without open (*PIO*) and close (*PIC*) delimiters.

To retrieve *processing instructions* for a book component (a document that is part of a FrameMaker book file), call `Sw_GetBookCompPi()`.

Returns

The *PI* string, or `NULL` on error. On error, `SRW_errno` is set to one of the following error codes:

- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_NOT_BOOK_COMP`

Examples

The following code retrieves the *PI* for the current conversion object if it is of type SW_OBJ_BOOK:

```
. . .
if(Sw_GetObjType(swObj) == SW_OBJ_BOOK) {
    StringT bookPI = Sw_GetBookPi(swObj);
    . . .
    F_ApiDeallocateString(&bookPI);
}
. . .
```

See also

- [“Sw_SetBookPi\(\)” on page 352](#)
- [“Sw_GetBookCompPi\(\)” on page 296](#)
- [“SwConvObjT” on page 424](#)
- [“SRW_erno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on StringT

Sw_GetChildConvObj()

Returns the export conversion object that is immediately subordinate to the specified object.

Synopsis

```
#include "fm_struct.h"
. . .
SwConvObjT Sw_GetChildConvObj(SwConvObjT parentObj);
```

Arguments

parentObj Parent conversion object to query

Details

As export events occur, corresponding conversion objects are created. These conversion object/event pairs are pushed onto a stack that represents the hierarchy of currently open conversion objects. Note that for container objects, import events come in pairs; for every BEGIN event to open a container there is a corresponding END event to close it. A conversion object is opened when the BEGIN event occurs, and is closed when the END event occurs.

Note that any number of nested conversion objects can be open at one time. However, among sibling objects only one sibling can be open at a time, because an object is removed from the stack as soon as it is closed.

```
<LEVEL_ONE>
  <LEVEL_TWO>
    <LEVEL_THREE>
      <LEVEL_FOUR>
        <LEVEL_FIVE></LEVEL_FIVE> (to be removed from stack)
```

Notice that LEVEL_FIVE has just posted its closing object, and will be removed from the stack. The next conversion object could easily be another LEVEL_FIVE object, but there will never be two open LEVEL_FIVE objects on the stack at the same time.

`Sw_GetChildConvObj()` returns the ID of the the open conversion object that is a child of the specified object. For example, starting from the current object of type `SW_OBJ_DOC`, you can use this function to get the highest level element in the document.

Returns

A conversion object, or `NULL` under the following conditions:

- No child object is associated with the specified conversion object.
- Child objects are associated with the specified conversion object, but no child object is open.

Examples

The following code finds the highest level element in the currently open FrameMaker document, and prints out its name:

```
. . .
SrwErrorT Sw_EventHandler(eventp, swObj)
SwEventT *eventp;
SwConvObjT swObj;
{
    SwConvObjT elemObj, docObj;
    SwEventT *elemEvtp;
    StringT s;
```

```
    if(eventp->evtype == SW_EVT_BEGIN_ELEM) {
        if (docObj = Sw_GetCurConvObjOfType(SW_OBJ_DOC)) {
            elemObj = Sw_GetChildConvObj(docObj);
            while (Sw_GetObjType(elemObj) != SW_OBJ_ELEM) {
                elemObj = Sw_GetChildConvObj(elemObj);
            }
            elemEvp = Sw_GetAssociatedEvent(elemObj);
            rootId = elemEvp->fm_objid;
            elemDefId = F_ApiGetId(docId, rootId, FP_ElementDef);
            s = F_ApiGetString(docId, elemDefId, FP_Name);
            F_Printf(NULL, "\nElement Name: %s", s);
            F_ApiDeallocateString(&s);
        }
    }
    return (Sw_Convert(eventp, swObj));
}
```

See also

- [“Sw_GetCurConvObj\(\)” on page 302](#)
- [“Sw_GetParentConvObj\(\)” on page 318](#)
- [“SwConvObjT” on page 424](#)

Sw_GetColSpecs()

Retrieves a list of *CALS* column specifications (*colspecs*) from a table export conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
SrwColSpecsT Sw_GetColSpecs(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

To retrieve the list of *colspecs* from a table associated with an export conversion object, call `Sw_GetColSpecs()`. Use this function only for exporting *CALS* tables. *swObj* must be an of type `SW_OBJ_TABLE`.

Important: On export, *colspecs* are not produced for table parts (table heading, table footing, table body).

After it is no longer needed, the `SrwColSpecsT` structure returned by this function should be deallocated with `Srw_DeallocateColSpecs()`.

Before conversion, you can set or change the list of *colspecs* for a table associated with an export conversion object by calling `Sw_SetColSpecs()`.

Returns

An `SrwColSpecsT` structure, or a zeroed-out structure on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code demonstrates a call to `Sw_GetColSpecs()`. For another example, see [“Sw_SetColSpecs\(\)” on page 354](#).

```
. . .
SrwErrorT Sw_EventHandler(SwEventT *eventp, SwConvObjT swObj)
{
    SrwColSpecsT colspecs;
    SwConvObjT tblObj;
    switch(Sw_GetObjType(swObj))
    {
        case SW_OBJ_TABLE_ROW:
            /* Retrieve current table object. */
            tblObj = Sw_GetCurConvObjOfType(SW_OBJ_TABLE);
            /* Retrieve copy of colspecs for table object. */
            colspecs = Sw_GetColSpecs(tblObj);
    }
    . . .
```

See also

- [“Sw_SetColSpecs\(\)” on page 354](#)
- [“Srw_DeallocateColSpecs\(\)” on page 251](#)
- [“SwConvObjT” on page 424](#)
- [“SrwColSpecsT” on page 419](#)
- [“SRW_errno” on page 418](#)

Sw_GetCurConvObj()

Retrieves the conversion object most recently passed to the export event handler.

Synopsis

```
#include "fm_struct.h"

. . .

SwConvObjT Sw_GetCurConvObj(VoidT);
```

Arguments

None.

Details

The current object is the one associated with the export event that is currently being processed by the export event handler. The current conversion object is a primary entry into the conversion object stack.

Because event handlers are passed the current object as a parameter, this function is generally reserved for use by procedures that don't get the current object as a parameter.

Given a conversion object, you can get information about its type, its associated event, and its parent or child object (if a child object exists).

Returns

The conversion object most recently passed to the event handler.

Examples

The following code might be found in a function that wasn't passed the current object or the current event. It retrieves the current conversion object from the stack, then gets the object type, and if the object is an element it processes the element in some way.

```
. . .
SwConvObjT swObj;
SwEventT *eventp;
SwObjTypeT swObjType;
F_ObjHandleT elemId;
. . .
swObj = Sw_GetCurConvObj();
if (Sw_GetObjType(swObj) == SR_OBJ_ELEM) {
    /* Process the element in some way */
    . . .
}
```

See also

- [“Sw_GetChildConvObj\(\)” on page 298](#)
- [“Sw_GetParentConvObj\(\)” on page 318](#)

- [“Sw_EventHandler\(\)” on page 286](#)
- [“SwConvObjT” on page 424](#)

Sw_GetCurConvObjOfType()

Retrieves the most recently opened export conversion object of a specific type.

Synopsis

```
#include "fm_struct.h"

. . .

SwConvObjT Sw_GetCurConvObjOfType(SwObjTypeT objType);
```

Arguments

objType Type of conversion object for which to search

Details

This function checks the conversion object stack, starting from the current object, for the nearest open conversion object of the specified type. Consider the following example:

```
<SESSION>
  <DOC>
    <ELEM (Section)>
      <ELEM (Para)>
        <XREF>
```

The current conversion object is the XREF. Two elements are open; one for the section and one for a paragraph in the section. The current object of type ELEM is the Para element because it is the nearest open object of that type.

A common use for this function is to retrieve the conversion object for the FrameMaker document out of which you are exporting markup. For example, you might need to get the document's ID, available in the document's conversion object, because you need to pass it as an argument to an FDK function or to another structure import/export API function.

Returns

The conversion object, or NULL if there is no object that matches the specified type of conversion object. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code retrieves the current document conversion object from the stack. It then gets the FrameMaker ID for that document.

Note the use of a separate variable for the document object. In the event handler, you should never clobber change the value of the current conversion object variable, because

the handler will almost certainly need that information to convert the current conversion object.

```
. . .
SwConvObjT docObj;
F_ObjHandleT docId;
. . .
docObj = Sw_GetCurConvObjOfType(SW_OBJ_TABLE);
if (docObj) {
    docId = Sw_GetDocId(docObj);
    /*Now you have the ID of the currently open document.*/
    . . .
}
```

See also

- [“Sw_GetCurConvObj\(\)” on page 302](#)
- [“Sw_GetChildConvObj\(\)” on page 298](#)
- [“Sw_GetParentConvObj\(\)” on page 318](#)
- [“SwConvObjT” on page 424](#)
- [“SwObjTypeT” on page 407](#)
- [“SRW_erno” on page 418](#)

Sw_GetDocId()

Retrieves the ID for the FrameMaker document file that is associated with the specified export conversion object. Note that the conversion object must be one of type SW_OBJ_DOC.

Synopsis

```
#include "fm_struct.h"
. . .
F_ObjHandleT Sw_GetDocId(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

This function retrieves the ID of a FrameMaker document.

Some FDK and structure import/export API functions require a FrameMaker document ID as an argument. Note that the ID doesn't exist until the document is opened for export. This means an event of type SW_EVT_BEGIN_DOC must have occurred, and the corresponding conversion object must have been converted. If you call Sw_GetDocId() before the specified object is converted, this function returns NULL.

Returns

The FrameMaker document ID, or NULL on error. On error, SRW_errno is set to SRW_E_WRONG_OBJ_TYPE.

Examples

The following code traps a document import event, and then retrieves the associated document ID.

```
. . .
SrErrorT Sw_EventHandler(eventp, swObj)
SwEventT *eventp;
SwConvObjT swObj;
{
    F_ObjHandleT docId;

    if ((eventp->evtype) == SW_EVT_BEGIN_DOC) {
        if (Sw_GetObjType(swObj) == SW_OBJ_DOC) {
            Sw_Convert(eventp, swObj);
            docId = Sw_GetDocId(swObj);
            /*Process the document in some way*/
        }
    }
    . . .
}
```

See also

- [“SwConvObjT” on page 424](#)
- [“Primitive data types” on page 383](#) for more information about F_ObjHandleT

Sw_GetEntityName()

Retrieves the name of the markup entity proposed by the specified export conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

StringT Sw_GetEntityName(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

To retrieve the name of an entity in markup proposed by an export conversion object, call `Sw_GetEntityName()`. To set an entity name for an export conversion object before it is converted, call `Sw_SetEntityName()`.

Returns

The name of the proposed entity, or NULL on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code prints the name of every proposed entity:

```
. . .
SrwrErrorT Sw_EventHandler(eventp, swObj)
    SwEventT *eventp;
    SwConvObjT swObj;
{
    StringT s;

    if(Sw_GetObjType(swObj) == SW_OBJ_ENTITY) {
        s = Sw_GetEntityName(swObj);
        F_Printf(NULL, "\nEnt Name Is: %s", s);
        F_ApiDeallocateString(&s);
    }
    return (Sw_Convert(eventp, swObj));
}
. . .
```

See also

- [“Sw_SetEntityName\(\)” on page 356](#)
- [“SwConvObjT” on page 424](#)
- [“SRW_errno” on page 418](#)

- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_GetExportFileFormat()

Retrieves the file format to use to export a graphic or equation associated with an export conversion object of type `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION`.

Synopsis

```
#include "fm_struct.h"

. . .

StringT Sw_GetExportFileFormat (SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

Call `Sw_GetExportFileFormat()` to retrieve the string identifying the file format proposed for exporting a graphic or equation associated with an `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION` conversion object.

In the absence of an explicit read/write rule or call to `Sw_SetExportFileFormat()` that specifies a file format for export of a graphic or equation, `Sw_GetExportFileFormat()` always returns `NULL`.

The following table describes how FrameMaker processes a graphic or equation conversion object based on the file format information and the number of objects in the anchored frame:

File Format	Number of objects	Strategy
<code>NULL</code>	1	FrameMaker examines the object's inset facet for a file format other than <code>FrameVector</code> or <code>FramedImage</code> . If a recognizable file format is found, the object is exported as is, without invoking a filter. Otherwise, FrameMaker invokes the CGM filter to export the object.
<code>NULL</code>	> 1	FrameMaker invokes the CGM filter to export the entire anchored frame.
<code>non-NULL</code>	1	FrameMaker examines the object's inset facet for a matching file format. If a matching format is found, the object is exported as is, without invoking a filter. Otherwise, FrameMaker attempts to convert the object by invoking a filter for the specified file format.
<code>non-NULL</code>	> 1	FrameMaker attempts to convert the entire anchored frame by invoking a filter for the specified file format.

If you specify a file format with `Sw_SetExportFileFormat()`, choose a format for which you have the necessary filter.

Returns

A string containing the file format or `NULL`.

Examples

The following code prints out the export file format for every conversion object of type `SW_OBJ_GRAPHIC`:

```
. . .
SrwErrorT Sw_EventHandler(eventp, swObj)
    SwEventT *eventp;
    SwConvObjT swObj;
{
    StringT s;

    if(Sw_GetObjType(swObj) == SW_OBJ_GRAPHIC) {
        s = Sw_GetExportFileFormat(swObj);
        F_Printf(NULL, "\nEnt Name Is: %s",
            F_ApiDeallocateString(&s);
    }
    return (Sw_Convert(eventp, swObj));
}
. . .
```

See also

- [“Sw_SetExportFileFormat\(\)” on page 357](#)
- [“SwConvObjT” on page 424](#)
- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_GetExportFilePath()

Retrieves the filepath intended for use when exporting a graphic or equation associated with an export conversion object to a separate file.

Synopsis

```
#include "fm_struct.h"

. . .

FilePathT *Sw_GetExportFilePath(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

To retrieve the proposed export filepath for a graphic or equation associated with an `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION` export conversion object, call `Sw_GetExportFilePath()`.

If `FilePathT` is not `NULL`, the graphic or equation associated with the export conversion object is exported to a separate file specified in `FilePathT`, and that file is referenced in the document instance. If `FilePathT` is `NULL`, the graphic or equation is not exported to a separate file.

`FilePathT` is a pointer to a platform-independent representation of a platform-specific path. For more information about working with platform-independent representations of filepaths, see Chapter 16, "Making I/O and Memory Calls Portable," in the *FDK Programmer's Guide*.

To set the filepath for exporting a graphic or equation to a standalone file before conversion, call `Sw_SetExportFilePath()`. To determine the file format for exporting a graphic or equation, call `Sw_GetExportFileFormat()`.

Returns

The filepath to be used for exporting a graphic or equation to a standalone file, or `NULL` if a separate graphic or equation file will not be created.

Examples

The following code retrieves the proposed filepath to use for exporting a graphic or equation associated with the current conversion object:

```
. . .
FilePathT *exportFilePath;
. . .
exportFilePath = Sw_GetExportFilePath(swObj);
. . .
F_FilePathFree(exportFilePath);
. . .
```

See also

- [“Sw_SetExportFilePath\(\)” on page 359](#)
- [“Sw_GetExportFileFormat\(\)” on page 307](#)
- [“SwConvObjT” on page 424](#)
- [“FilePathT” on page 409](#)

Sw_GetGfxDataAttrVals()

Retrieves the list of data attributes for a graphics or equation export conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
StructuredAttrValsT Sw_GetGfxDataAttrVals(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

To retrieve a list of data attributes for a graphic or equation associated with an `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION` export conversion object, call `Sw_GetGfxDataAttrVals()`.

Important: When you no longer need the attribute list returned by this function, delete it and free the memory allocated for it with a call to `Structured_DeallocateAttrVals()`.

To assign a list of data attributes to a graphics or equation export conversion object, call `Sw_SetGfxDataAttrVals()`.

Returns

A structure containing a list of pointers to attribute values, or a zeroed-out structure on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

For more detailed examples of handling attribute values, see [“Sw_GetAttrVal\(\)” on page 290](#), [“Sw_GetAttrVals\(\)” on page 293](#), [“Sw_SetAttrVal\(\)” on page 345](#), and

[“Sw_SetAttrVals\(\)” on page 347](#). The following code retrieves the list of data attributes for the current conversion object of type `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION`:

```
. . .
StructuredAttrValsT attributes;
. . .
attributes = Sw_GetGfxDataAttrVals(swObj);
. . .
```

See also

- [“Sw_SetGfxDataAttrVals\(\)” on page 360](#)
- [“SwConvObjT” on page 424](#)
- [“StructuredAttrValsT” on page 413](#)
- [“SRW_errno” on page 418](#)

Sw_GetGfxEntityName()

Retrieves the entity name for the graphic or equation associated with the specified export conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
StringT Sw_GetGfxEntityName(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

To determine the proposed entity name for a graphic or equation associated with a specific export conversion object of type `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION`, call `Sw_GetGfxEntityName()`.

To set or change the entity name for a graphic or equation before converting it, call `Sw_SetGfxEntityName()`.

To determine a graphic entity's type, call `Sw_GetGfxEntityType()`.

Returns

The entity name, or `NULL` on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code retrieves the proposed entity name for the graphic or equation associated with an `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION` conversion object:

```
. . .
StringT entityName;
. . .
if((Sw_GetObjType(swObj) == SW_OBJ_GRAPHIC) ||
    (Sw_GetObjType(swObj) == SW_OBJ_EQUATION)) {
    entityName = Sw_GetGfxEntityName(swObj);
    . . .
    F_ApiDeallocateString(&entityName);
}
. . .
```

See also

- [“Sw_SetGfxEntityName\(\)” on page 361](#)
- [“Sw_GetGfxEntityType\(\)” on page 312](#)
- [“SwConvObjT” on page 424](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_GetGfxEntityType()

Retrieves the entity type for the graphic or equation associated with the specified export conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
StructuredEntityTypeT Sw_GetGfxEntityType(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

To determine the type of markup entity assigned to a graphic or equation associated with an `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION` export conversion object, call `Sw_GetGfxEntityType()`. The entity type returned by this function is *CDATA*, *SDATA*, or *NDATA*.

To specify the entity type for a graphic or equation export conversion object before it is converted, call `Sw_SetGfxEntityType()`.

To determine the name of the entity associated with a graphic or equation export conversion object, call `Sw_GetGfxEntityName()`.

Returns

One of the following enumerated types corresponding to markup entity types, or `NULL` on error:

- `STRUCTURED_ET_CDATA`, if the entity type is *CDATA*.
- `STRUCTURED_ET_SDATA`, if the entity type is *SDATA*.
- `STRUCTURED_ET_NDATA`, if the entity type is *NDATA*.

On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code retrieves the entity type for the current conversion object if it is of type `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION`:

```
. . .
StructuredEntityTypeT entityType;
. . .
if((Sw_GetObjType(swObj) == SW_OBJ_GRAPHIC) ||
    (Sw_GetObjType(swObj) == SW_OBJ_EQUATION))
    entityType = Sw_GetGfxEntityType(swObj);
. . .
```

See also

- [“Sw_SetGfxEntityType\(\)” on page 362](#)
- [“Sw_GetGfxEntityName\(\)” on page 311](#)
- [“SwConvObjT” on page 424](#)
- [“SRW_errno” on page 418](#)

Sw_GetGfxNotation()

Retrieves the *notation name* for the specified graphic or equation export conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

StringT Sw_GetGfxNotation(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

Call `Sw_GetGfxNotation()` to retrieve the proposed name of the *notation* for a graphic or equation associated with an `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION` export conversion object.

To specify a *notation name* for a graphic or equation conversion object before it is converted, call `Sw_SetGfxNotation()`.

Returns

A string containing the *notation name*, or `NULL` on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code retrieves the *notation name* for the current conversion object if it is of type `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION`:

```
. . .
StringT notationName;

. . .
if((Sw_GetObjType(swObj) == SW_OBJ_GRAPHIC) ||
    Sw_GetObjType(swObj) == SW_OBJ_EQUATION))
    notationName = Sw_GetGfxNotation(swObj);

. . .
F_ApiDeallocateString(&notationName);
```

See also

- [“Sw_SetGfxNotation\(\)” on page 364](#)
- [“SwConvObjT” on page 424](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_GetGfxPubId()

Retrieves the *public identifier* for the graphic or equation associated with the specified export conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
StringT Sw_GetGfxPubId(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

Call `Sw_GetGfxPubId()` to determine the *public identifier* for a graphic or equation associated with a specific export conversion object of type `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION`.

To set or change the *public identifier* for a graphic or equation export conversion object before it is converted, call `Sw_SetGfxPubId()`. To determine the *system identifier* for a graphic or equation export conversion object, call `Sw_GetGfxSysId()`.

Returns

A string containing the *public identifier*, or `NULL` on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code retrieves the *public identifier* for a conversion object that is of type `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION`:

```
. . .
StringT publicIdentifier;
. . .
publicIdentifier = Sw_GetGfxPubId(swObj);
. . .
F_ApiDeallocateString(&publicIdentifier);
. . .
```

See also

- [“Sw_SetGfxPubId\(\)” on page 365](#)
- [“Sw_GetGfxSysId\(\)” on page 316](#)
- [“SwConvObjT” on page 424](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_GetGfxSysId()

Retrieves the *system identifier* for the graphic or equation associated with the specified export conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
StringT Sw_GetGfxSysId(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

Call `Sw_GetGfxSysId()` to determine the *system identifier* for a graphic or equation associated with a specific export conversion object of type `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION`.

To set or change the *system identifier* for a graphic or equation export conversion object before it is converted, call `Sw_SetGfxSysId()`. To determine the *public identifier* for a graphic or equation export conversion object, call `Sw_GetGfxPubId()`.

Returns

A string containing the *system identifier*, or `NULL` on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code retrieves the *system identifier* for a graphic associated with a specific conversion object:

```
. . .
StringT systemIdentifier;
. . .
systemIdentifier = Sw_GetGfxSysId(swObj);
. . .
F_ApiDeallocateString(&systemIdentifier);
```

See also

- [“Sw_SetGfxSysId\(\)” on page 366](#)
- [“Sw_GetGfxPubId\(\)” on page 315](#)
- [“SwConvObjT” on page 424](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_GetObjType()

Retrieves the object type of the specified conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

SwObjTypeT Sw_GetObjType(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

This function is commonly used to test the a conversion object's type in an event handler in order to determine how to process the object before converting it. The conversion object types are defined by the `SwObjTypeT` data type. See [“SwObjTypeT” on page 407](#) for the list of object types.

Note that you cannot modify a conversion object's type.

Returns

The object type proposed by the specified conversion object.

Examples

The following code tests the current conversion object to see if it is of type `SW_OBJ_BOOK`. If it is, code inside the `if` statement is executed.

```
. . .
SrConvObjT swObj;
. . .
if (Sw_GetObjType(swObj) == SW_OBJ_BOOK) {
    /*You know the specified conversion object indicates a book*/
. . .
```

See also

- [“Sw_GetCurConvObjOfType\(\)” on page 303](#)
- [“Sw_GetCurConvObj\(\)” on page 302](#)
- [“SwConvObjT” on page 424](#)
- [“SwObjTypeT” on page 407](#)

Sw_GetParentConvObj()

Retrieves the export conversion object that is immediately superior to the specified export conversion.

Synopsis

```
#include "fm_struct.h"

. . .

SwConvObjT Sw_GetParentConvObj(SwConvObjT childConvObj);
```

Arguments

childConvObj Conversion object to query

Details

As export events occur, corresponding conversion objects are created. These conversion object/event pairs are pushed onto a stack that represents the hierarchy of currently open conversion objects. Note that for container objects, import events come in pairs; for every BEGIN event to open a container there is a corresponding END event to close it. A conversion object is opened when the BEGIN event occurs, and is closed when the END event occurs.

Note that any number of nested conversion objects can be open at one time. However, among sibling objects only one sibling can be open at a time, because an object is removed from the stack as soon as it is closed.

```
<LEVEL_ONE>
  <LEVEL_TWO>
    <LEVEL_THREE>
      <LEVEL_FOUR>
```

`Sw_GetParentConvObj()` returns the ID of the the open conversion object that is immediately superior to the specified object. In the above example, if `LEVEL_THREE` is the specified object, `LEVEL_TWO` is its parent.

A use for this function might be to get the parent of a table. If the parent object is an element named `FINANCIAL_DATA`, you could apply a specific table format to the table.

In your event handler routines, you must declare a local variable of type `SwConvObjT` for the conversion object returned by this function.

Returns

The parent conversion object, or `NULL` if the specified conversion object is not subordinate to another conversion object.

Examples

The following code checks the current object type. If it is a table, it checks whether the parent object indicates a FINANCIAL_DATA element:

```
. . .
SwConvObjT swObj, containerObj;
StringT s;
. . .
if (Sw_GetObjType(swObj) == SW_OBJ_TABLE) {
    containerObj = Sw_GetParentConversionObj(swObj);
    s = Sw_GetStructuredGi(containerObj);
    if (F_StrIEqual(s, (StringT)"FINANCIAL_DATA"){
        /*Set the table format accordingly*/
        . . .
    }
    F_ApiDeallocateString(&s);
. . .
```

See also

- [“Sw_GetChildConvObj\(\)” on page 298](#)
- [“Sw_GetCurConvObj\(\)” on page 302](#)
- [“SwConvObjT” on page 424](#)

Sw_GetPI()

Retrieves the *processing instructions (PI)* associated with an export conversion object of type SW_OBJ_PI.

Synopsis

```
#include "fm_struct.h"
. . .
StringT Sw_GetPI(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

To fetch the *PI* for the specified export conversion object of type SW_OBJ_PI, call `Sw_GetPI()`.

To specify a different *PI* for a conversion object before it is converted, call `Sw_SetPI()`.

Returns

The *PI*, or NULL on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code retrieves the *PI* for the current conversion object of type `SW_OBJ_PI`:

```
. . .
SwConvObjT piObj;
StringT piStr;

. . .
piObj = Sw_GetCurConvObjOfType(SW_OBJ_PI);
piStr = Sw_GetPI(piObj);

. . .
F_ApiDeallocateString(&piStr);

. . .
```

See also

- [“Sw_SetPI\(\)” on page 367](#)
- [“SwConvObjT” on page 424](#)
- [“SRW_erno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_GetPrivateData()

Returns a pointer to data your application has associated with the specified conversion object. This pointer is the address of an arbitrary data type that you have allocated, initialized, and maintained.

Synopsis

```
#include "fm_struct.h"

. . .
PtrT Sw_GetPrivateData(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

You can allocate and maintain data that is private to your application, and then assign the address of that data to a specific conversion object. This is a way to save a state within your application, and then associate it with a specific level in the hierarchy of the conversion object stack. This function retrieves a pointer to such data that was previously set to the specified conversion object.

Note that your application owns the data. Also, the data can be of any addressable type, from a single variable to a structure within a deep hierarchy of structures. It is up to your application to maintain the data content.

Important: When the conversion object is closed, it is removed from the conversion object stack and its allocated memory is freed. However, the data referenced by the private data pointer is not deallocated. If the only reference to this data is stored with the conversion object, and that object is closed, you will not be able to deallocate the private data. Be sure to trap ending events (for example, `SR_EVT_END_ELEM`) and check their associated objects for private data pointers.

Returns

A pointer to private data, or `NULL` if no private data is set for the object.

Examples

The following code declares a structure that will be used to keep data about tables. It initializes the structure, and uses `Sw_SetPrivateData()` to assign the structure to the a table.

Then for each row, the code increments the row counter for that table. Before closing the table object, the code prints out the number of table rows, and then deallocates the private data.

```
. . .
typedef struct {
    IntT rowCount;
    BoolT isFinancial;
} MyDataT;

SrwErrorT Sr_EventHandler(eventp, srObj)
SwEventT *eventp;
SwConvObjT swObj;
{
    SwEventT *parentEvent;
    SwConvObjT parentObj;
    MyDataT *testData;

    case SW_EVT_BEGIN_TABLE:
        testData = F_Alloc(sizeof(MyDataT), NO_DSE);
        testData->rowCount = (IntT)0;
        testData->isFinancial = (BoolT)True;
        Sw_SetPrivateData(swObj, testData);
        break;
```

```
    case SW_EVT_END_TABLE:
        testData = Sw_GetPrivateData(swObj);
        F_Printf(NULL, "\nTable has %d rows",
                                testData->rowCount);

        F_Free(testData);
        Sw_SetPrivateData(swObj, 0);
        break;
    case SW_EVT_BEGIN_TABLE_ROW:
        parentObj = Sw_GetParentConvObj(swObj);
        parentEvent = Sw_GetAssociatedEvent(parentObj);
        while(!(F_StrIEqual(Sw_GetStructuredGi(parentObj),
                                "TABLE"))) {
            parentObj = Sw_GetParentConvObj(parentObj);
            parentEvent = Sw_GetAssociatedEvent(parentObj);
        }
        testData = Sw_GetPrivateData(parentObj);
        testData->rowCount++;
        break;
    }
    return (Sw_Convert(eventp, swObj));
}
```

See also

- [“Sw_SetPrivateData\(\)” on page 368](#)
- [“SwConvObjT” on page 424](#)
- [“Primitive data types” on page 383](#) for more information on PtrT

Sw_GetProcessingFlags()

Returns the status of the element processing flags that indicate the processing for an export conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

IntT Sw_GetProcessingFlags(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

If you call this function before an object is converted, it reports the proposed processing for the object. You can modify this proposal by calling `Sw_SetProcessingFlags()`. Note that if the conversion object is not for a type of element, this function returns 0.

If you call this function after an object is converted, it reports the processing that was performed on the object.

Returns

A status integer set to one of the following values:

- 0
This indicates either the object will be converted normally, or it is not an element.
- `SRW_UNWRAP_ELEMENT`
This indicates the `unwrap` read/write rule was specified for the given element.
- `SRW_DROP_ELEMENT_CONTENT`
This indicates the `drop content` read/write rule was specified for the given element.
- `SRW_UNWRAP_ELEMENT | SRW_DROP_ELEMENT_CONTENT`
This indicates both the `unwrap` and `drop content` read/write rules were specified for the given element.

Examples

The following code gets the processing flags for each element and prints out what their effect will be.

```
. . .

SrwErrorT Sw_EventHandler(eventp, swObj)
SwEventT *eventp;
SwConvObjT swObj;
```

```
{
    switch(Sw_GetProcessingFlags(swObj)) {
    case SRW_DROP_ELEMENT_CONTENT:
        F_Printf(NULL, "\nDrop Content of %s",eventp->u.tag.gi);
        break;
    case SRW_UNWRAP_ELEMENT:
        F_Printf(NULL, "\nUnwrap %s",eventp->u.tag.gi);
        break;
    case SRW_DROP_ELEMENT | SRW_UNWRAP_ELEMENT:
        F_Printf(NULL, "\nEliminate %s",eventp->u.tag.gi);
        break;
    }
    return (Sw_Convert(eventp, swObj));
}
```

See also

- [“Sw_SetProcessingFlags\(\)” on page 369](#)
- [“SwConvObjT” on page 424](#)
- [“Primitive data types” on page 383](#) for more information on `IntT`

Sw_GetPropVal()

Retrieves a specified FrameMaker property value for an export conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
StringT Sw_GetPropVal(SwConvObjT swObj,
    SrwFmPropertyT fmProp);
```

Arguments

<i>swObj</i>	Conversion object to query
<i>fmProp</i>	Property structure to examine

Details

Conversion objects can have properties associated with them. The properties that can be assigned to an object depend on the conversion object type. The objects that can have properties are of the following types:

SW_OBJ_ELEM	SW_OBJ_TABLE	SW_OBJ_TABLE_HEADING
SW_OBJ_EQUATION	SW_OBJ_TABLE_BODY	SW_OBJ_TABLE_ROW
SW_OBJ_FOOTNOTE	SW_OBJ_TABLE_CELL	SW_OBJ_TABLE_TITLE

SW_OBJ_GRAPHIC	SW_OBJ_TABLE_COLSPEC	SW_OBJ_VARIABLE
SW_OBJ_MARKER	SW_OBJ_TABLE_FOOTING	SW_OBJ_XREF

For a list of the properties that apply to these object types, see [“SrwFmPropertyT” on page 397](#).

Note that the property value is a string, in spite of the fact that some of the properties are enumerated types or other numeric values. These numeric values are handled as strings to unify the format of all conversion object properties.

Important: When the string returned by this function is no longer needed, call `F_StrFree()` to free it.

If you call this function before an object is converted, it reports the proposed value for the property. You can change this value with `Sw_SetPropVal()`. If you call it after an object is converted, it reports the value used for conversion. Once converted, this value cannot be changed.

To retrieve a list of all property value data structures for an export conversion object, call `Sw_GetPropVals()`.

Returns

A string containing the value of the specified property, or `NULL` on error. On error, `SRW_errno` is set to one of the following error codes:

- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_BAD_VALUE`
- `SRW_E_OBJ_HAS_NO_SUCH_PROP`

Examples

The following code traps a table event and gets the table format property value. The code then uses a property to set the CALS column specification, and frees the string that stores the table's format property value.

```
SrwErrorT Sr_EventHandler(eventp, swObj)
SwEventT *eventp;
SwConvObjT swObj;
{
    StringT tabFmt;
```

```
    if ((eventp->evtype) == SW_EVT_BEGIN_TABLE) {
        tabFmt = Sw_GetPropVal(swObj, SRW_PROP_TABLE_FORMAT);
        if (F_StrIEqual(tabFmt, (StringT) "Financial Table"))
            Sw_SetPropVal(swObj, SRW_PROP_COL_NAME,
                (StringT) "FINANCIAL");
        F_StrFree(tabFmt);
        . . .
    }
    . . .
}
```

See also

- [“Sw_GetPropVals\(\)” on page 326](#)
- [“Srw_DeallocatePropVal\(\)” on page 252](#)
- [“SwConvObjT” on page 424](#)
- [“SrwFmPropertyT” on page 397](#)
- [“SRW_erno” on page 418](#)

Sw_GetPropVals()

Retrieves a list of properties for the specified export conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
SrwPropValsT Sw_GetPropVals(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

Conversion objects can have properties associated with them. The properties that can be assigned to an object depend on the conversion object type. The objects that can have properties are of the following types:

SW_OBJ_ELEM	SW_OBJ_TABLE	SW_OBJ_TABLE_HEADING
SW_OBJ_EQUATION	SW_OBJ_TABLE_BODY	SW_OBJ_TABLE_ROW
SW_OBJ_FOOTNOTE	SW_OBJ_TABLE_CELL	SW_OBJ_TABLE_TITLE
SW_OBJ_GRAPHIC	SW_OBJ_TABLE_COLSPEC	SW_OBJ_VARIABLE
SW_OBJ_MARKER	SW_OBJ_TABLE_FOOTING	SW_OBJ_XREF

For a list of the properties that apply to these object types, see [“SrwFmPropertyT” on page 397](#).

This function returns `SrwPropValsT`, which is a list of all the properties for a specified conversion object. `SrwPropValsT` is defined as:

```
typedef struct {
    IntT len; /* Number of properties in array. */
    SrwPropValT *val; /* Pointer to array of values. */
} SrwPropValsT;
```

Each property is expressed by a `SrwPropValT` structure, which is defined as:

```
typedef struct {
    SrwFmPropertyT prop;
    StringT value;
} SrwPropValT;
```

Note that the property value is a string, in spite of the fact that some of the properties are enumerated types or other numeric values. These numeric values are handled as strings to unify the format of all conversion object properties.

Important: Note that actual property values are strings. If you want to store a value in a variable, then you must either use `Sw_GetPropVal()`, or else use `F_StrCopyString()`, as follows:

```
propVal = F_StrCopyString(propVals.vals[i].value).
```

Important: When you are finished with the list of property values, be sure to call `Srw_DeallocatePropVals()` to free its memory. Use `Srw_DeallocatePropVal()` to free up a single property value.

If you call this function before an object is converted, it reports the proposed list of property values to use for conversion. If you call this function after the object was converted, it returns the list of properties that was used. To specify a different list of property values for an object before it is converted, call `Sw_SetPropVals()`. You cannot change the property values after the object is converted.

To retrieve the string that represents a single property value, call `Sw_GetPropVal()`.

Returns

The property values list associated with the conversion object, or `NULL` on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code traps a table object, gets its list of properties, and loops through the list.

```
SrwErrorT Sr_EventHandler(eventp, swObj)
SwEventT *eventp;
SwConvObjT swObj;
```

```
{
    SrwPropValsT propVals;
    StringT propVal;
    UIntT i;

    if ((eventp->evtype) == SW_EVT_BEGIN_TABLE) {
        propVals = Sw_GetPropVals(swObj);
        for (i = 0; i < propVals.len; i++) {
            switch (propVal.val[i].prop) {
                case SRW_PROP_COLUMNS :
                    /* Note that the property value is a string! */
                    propVal = F_StrCopyString(propVals.val[i].value);
                    /* Process swObj */
                    . . .
                    break;
                case SRW_PROP_COLUMN_WIDTHS :
                    /* Process swObj */
                    . . .
                    break;
            }
            . . .
        }
        Srw_DeallocatePropVals(&propVals);
    }
    . . .
}
```

See also

- [“Sw_GetPropVal\(\)” on page 324](#)
- [“Srw_DeallocatePropVals\(\)” on page 253](#)
- [“SwConvObjT” on page 424](#)
- [“SrwPropValsT” on page 421](#)
- [“SRW_erno” on page 418](#)

Sw_GetSessionProps()

Retrieves export session properties for the current session (as specified by a conversion object of type `SW_OBJ_SESSION`).

Synopsis

```
#include "fm_struct.h"

. . .

SwSessionPropsT Sw_GetSessionProps(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

To retrieve the current session properties, you should first call `Sw_GetCurConvObjOfType(SW_OBJ_SESSION)` to get the Id of the current session object, then pass that Id to `Sw_GetSessionProps()`.

To set or change import session properties, call `Sw_SetSessionProps()`.

Important: Be sure to call `Sw_DeallocateSessionProps()` to deallocate the `SwSessionPropsT` returned by this call when the structure is no longer needed.

Returns

An `SwSessionPropsT` structure describing current import session property settings.

`SwSessionPropsT` is defined as:

```
typedef struct {
    BoolT includeDtd;
    BoolT includeSgmlDecl;
    StringT doctypeSysId;
    stringT doctypePubId;
    BoolT overWriteFiles; /*True if batch can overwrite files of
                           same name in the target directory*/
} SwSessionPropsT;
```

Examples

The following code gets the Id for the current session conversion object, then retrieves the session properties for export FrameMaker documents. Finally, the code frees the space allocated to the `SwSessionPropsT` structure:

```
. . .
SwSessionPropsT exportSessionProps;
SwConvObjT sessionObj;
. . .
```

```
sessionObj = SwGetCurConvObjOfType(SW_OBJ_SESSION);
exportSessionProps = Sw_GetSessionProps(sessionObj);
. . .
Sw_DeallocateSessionProps(&exportSessionProps);
. . .
```

See also

- [“Sw_SetSessionProps\(\)” on page 370](#)
- [“Sw_DeallocateSessionProps\(\)” on page 284](#)
- [“Sw_GetCurConvObjOfType\(\)” on page 303](#)
- [“SwConvObjT” on page 424](#)
- [“SwSessionPropsT” on page 425](#)

Sw_GetStructuredGi()

Deprecated: Sw_GetSgmlGi()

Retrieves the markup generic identifier (GI) associated with the specified export conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
StringT Sw_GetStructuredGi(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

This function retrieves the markup GI associated with the specified conversion object. To have an associated GI, *swObj* must be one of the following object types:

SW_OBJ_ELEM	SW_OBJ_TABLE_FOOTING	SW_OBJ_EQUATION
SW_OBJ_TABLE	SW_OBJ_TABLE_ROW	SW_OBJ_VARIABLE
SW_OBJ_TABLE_TITLE	SW_OBJ_TABLE_CELL	SW_OBJ_MARKER
SW_OBJ_TABLE_HEADING	SW_OBJ_TABLE_COLSPEC	SW_OBJ_XREF
SW_OBJ_TABLE_BODY	SW_OBJ_GRAPHIC	SW_OBJ_FOTNOTE

The read/write rules for your current structure application map the exported FrameMaker element to an element in markup of a specific type, which determines the GI for the exported element. In most cases, this mapping is sufficient. You can use this function to check the mapping before converting the object, and then change the GI if necessary.

This function is also an easy way to retrieve the GI associated with any open conversion object. For example, you can get the GI of the highest level element associated with the FrameMaker document you are exporting.

To change the GI before converting an object, call `Sw_SetStructuredGi()`.

Returns

A string containing the GI, or NULL on error. On error, `SRW_errno` is set to `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code retrieves the markup GI corresponding to the highest level element in the FrameMaker document. Note, the export event for that element must already have been opened for this export session.

```
. . .
SwConvObjT rootElemObj;
StringT elemGi;
. . .
rootElemObj = Sw_GetCurObjOfType(SW_OBJ_DOC);
rootElemObj = Sw_GetChildConvObj(rootElemObj);
while (Sw_GetObjType(rootElemObj) != SW_OBJ_ELEM) {
    rootElemObj = Sw_GetChildConvObj(rootElemObj)
}
/* Now you can get the GI of the root element for the */
/* main flow of the currently open document. */
elemGi = Sw_GetStructuredGi(swObj);
. . .
F_ApiDeallocateString(&elemGi);
. . .
```

See also

- [“Sw_SetStructuredGi\(\)” on page 372](#)
- [“SwConvObjT” on page 424](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_GetStructuredText()

Deprecated: *Sw_GetSgmlText()*

Retrieves the FrameMaker text associated with a specified export conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

SwTextItemsT Sw_GetStructuredText(SwConvObjT swObj);
```

Arguments

swObj Conversion object to query

Details

Call *Sw_GetStructuredText()* to retrieve FrameMaker text that is associated with an export conversion object and that is proposed as *CDATA* for a markup document.

To set FrameMaker text before converting an object, call *Sw_SetStructuredText()*.

When it is no longer needed, call *Sw_DeallocateTextItems()* to delete the data structure returned by *Sw_GetStructuredText()* and free the memory allocated for it.

Returns

An *SwTextItemsT* structure containing the requested text, or *NULL* on error. On error, *SRW_errno* is set to *SRW_E_WRONG_OBJ_TYPE*.

Examples

The following code retrieves the text proposed for the current conversion object. If any text is proposed, it then prints out the GI for the currently open element, the length of the text items structure, and the integer for the event type. It then shows how you can print out the value for every text item. Finally, it deallocates the text items structure.

```
. . .
SrwErrorT Sw_EventHandler(eventp, swObj)
    SwEventT *eventp;
    SwConvObjT swObj;
{
    SwTextItemsT ti;
    UIntT i;
    SwConvObjT swElemObj;
    StringT elemGi;
```

```
ti = Sw_GetStructuredText(swObj);
if(ti.len > 0) {
    elemGi = Sw_GetStructuredGi(swElemObj);
    swElemObj = Sw_GetCurConvObjOfType(SW_OBJ_ELEM);
    F_Printf(NULL, "\n## %s ##", elemGi);
    F_ApiDeallocateString(&elemGi);
    F_Printf(NULL, "\nTI LEN: %d, event type: %d",
        ti.len, eventp->evtype);
    for(i=0;i<ti.len;i++) {
        switch(ti.val[i].itemType) {
            case SW_TEXT_STRING:
                F_Printf(NULL, "\nSW_TEXT_STRING: %s",
                    ti.val[i].u.text);
                break;
            case SW_TEXT_NUMERIC_CHAR_REF:
            case SW_TEXT_NAMED_CHAR_REF:
            case SW_TEXT_ENT_REF:
            default:
                break;
        } /* End of switch */
    }
    Sw_DeallocateTextItems(&ti);
}
return (Sw_Convert(eventp, swObj));
}
```

See also

- [“Sw_DeallocateTextItems\(\)” on page 285](#)
- [“Sw_SetStructuredText\(\)” on page 373](#)
- [“SwConvObjT” on page 424](#)
- [“SwTextItemsT” on page 426](#)
- [“SRW_erno” on page 418](#)

Sw_IsExportingToXml()

Tests whether FrameMaker is saving the current document as XML.

Synopsis

```
#include "fm_struct.h"

. . .

BoolT Sw_IsExportingToXml();
```

Arguments

none This function takes no arguments

Details

It is possible to create one import/export client that works for both XML and SGML. However, when the user saves a structured document as XML you might want to handle some of the elements differently than you would if the user saved the same document as SGML. Use this function to test whether the user wants XML or SGML output.

Returns

True if the the user is saving as XML.

Examples

The following code tests whether we are exporting to XML:

```
. . .
BoolT isXML;
. . .
if ((eventp->evtype) == SW_EVT_BEGIN_WRITER) {
    isXML = Sw_IsExportingToXml();
. . .
```

Sw_IsGeneralEntityDefined()

Tests for the existence of an *entity name declaration* in the DTD subset or in the *DOCTYPE* declaration.

Synopsis

```
#include "fm_struct.h"

. . .

BoolT Sw_IsGeneralEntityDefined(StringT ename);
```

Arguments

ename The entity name to test

Details

To determine if a entity name is declared in the DTD subset or in the *DOCTYPE* declaration, call `Sw_IsGeneralEntityDefined()`. On export, both an SGML declaration (for SGML) and a DTD must be provided as subsets of your application.

In a DTD, *ename* is case-sensitive according to the naming rules for SGML or XML. In an SGML declaration, *ename* is case-sensitive as specified in the reference concrete syntax.

Returns

`True` if the entity name is declared in the DTD subset or in the *DOCTYPE* declaration.
`False` if the entity name is not declared in either location.

Examples

The following code tests for the presence of an entity name in the DTD subset or *DOCTYPE* declaration:

```
. . .
BoolT isDefined;
StringT entityName;
. . .
if ((eventp->evtype) == SW_EVT_BEGIN_DOC) {
    isDefined = Sw_IsGeneralEntityDefined(entityName);
. . .
```

See also

- [“Sw_IsGeneralEntityNameUsed\(\)” on page 336](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` and `BoolT`

Sw_IsGeneralEntityNameUsed()

Tests an entity name to see if it was already used for exporting a graphic or equation in the DTD subset or in the *DOCTYPE* declaration.

Synopsis

```
#include "fm_struct.h"

. . .

BoolT Sw_IsGeneralEntityNameUsed(StringT ename);
```

Arguments

ename The entity name to test

Details

To determine if an entity name was already used to export a graphic or equation, call `Sw_IsGeneralEntityNameUsed()`.

Returns

True if the entity name is used; False if the entity name is not used.

Examples

The following code tests an entity name to see if was already used:

```
. . .
BoolT isUsed;
StringT entityName;

. . .
isUsed = Sw_IsGeneralEntityNameUsed(entityName);

. . .
```

See also

- [“Sw_IsGeneralEntityDefined\(\)” on page 335](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` and `BoolT`

Sw_NotifyEndTag()

Informs FrameMaker that an *end-tag* has been written to the markup document instance by a custom event handler.

Synopsis

```
#include "fm_struct.h"
. . .
VoidT Sw_NotifyEndTag(StringT gi);
```

Arguments

gi *Generic Identifier* for the markup element whose *end-tag* is written

Details

When your custom event handler writes an *end-tag* directly to the markup document instance instead of calling `Sw_Convert()` to perform the conversion, the client must call `Sw_NotifyEndTag()` to notify FrameMaker that an *end-tag* has been written to the markup instance for a given element.

gi is a string containing the *generic identifier (GI)* for the element whose *end-tag* is written. *gi* is case-sensitive according to XML standard or the reference concrete syntax of the SGML document declaration.

Returns

VoidT.

Examples

The following code writes an *end-tag* to the document instance, and notifies FrameMaker:

```
. . .
Sw_WriteDelimiter(SW_LOC_INSTANCE, STRUCTURED_DE_TAGC);
Sw_NotifyEndTag((StringT)"MEMO");
. . .
```

See also

- [“Sw_NotifyStartTag\(\)” on page 339](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` and `VoidT`

Sw_NotifyGeneralEntityDef()

Notifies FrameMaker that an *entity declaration* has been written to the DTD subset by a custom event handler.

Synopsis

```
#include "fm_struct.h"

. . .

VoidT Sw_NotifyGeneralEntityDef(StringT ename);
```

Arguments

ename Name of the *entity declaration* written

Details

To inform FrameMaker that an *entity declaration* is written to the DTD subset by your custom event handler, call `Sw_NotifyGeneralEntityDef()`. This function is only necessary when you write directly to the DTD instead of calling `Sw_Convert()`.

On export, a DTD must be provided as a subset of your application. In a DTD, *ename* is case-sensitive according to XML or SGML naming rules. In an SGML declaration, *ename* is case-sensitive as specified in the reference concrete syntax.

Returns

VoidT.

Examples

The following code informs FrameMaker that an *entity declaration* is written to the DTD subset:

```
. . .
StringT entityName;
. . .
Sw_NotifyGeneralEntityDef(entityName);
. . .
```

See also

- [“Sw_NotifyEndTag\(\)” on page 337](#)
- [“Sw_NotifyEndTag\(\)” on page 337](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` and `VoidT`

Sw_NotifyStartTag()

Notifies FrameMaker that a *start-tag* has been written to the markup document instance by a custom event handler.

Synopsis

```
#include "fm_struct.h"

. . .

VoidT Sw_NotifyStartTag(StringT gi);
```

Arguments

gi Generic identifier (*GI*) for the element whose *start-tag* is written

Details

To notify FrameMaker that your custom event handler wrote a *start-tag* to the markup document instance for a given element, call `Sw_NotifyStartTag()`. This function is only necessary when you write directly to the document instance instead of calling `Sw_Convert()`.

gi is a string containing the markup *generic identifier (GI)* for the element whose *start-tag* is written. *gi* is case-sensitive according to the XML standard or the reference concrete syntax of the SGML document declaration.

Returns

VoidT.

Examples

The following code notifies FrameMaker that a *start-tag* for an element has been written to the markup document instance:

```
. . .
StringT giText;
. . .
Sw_NotifyStartTag(giText);
. . .
```

See also

- [“Sw_NotifyEndTag\(\)” on page 337](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` and `VoidT`

Sw_ScanElem()

Processes a specified FrameMaker element outside of the order of sibling elements represented within its parent element. For example, if a document element contains `FrontMatter`, `Body`, and `EndMatter` elements, you can call `Sw_ScanElem()` to process `Body` ahead of `FrontMatter` and `EndMatter`.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sw_ScanElem(F_ObjHandleT docId, F_ObjHandleT elemId);
```

Arguments

<i>docId</i>	ID of the document containing the element to scan
<i>elemId</i>	ID of the element to scan

Details

It is possible that the order of elements in a markup document should be different than the order of elements in a FrameMaker document. For example, a glossary might be the first element in the markup file's `DOC` element, but you might want it last when the document is in FrameMaker. `Sw_ScanElem()` gives you the means to export elements to markup in any order.

In the example above, you would trap the event for the `DOC` element and then use `Sw_ScanElem()` to export the glossary first. You could then use `Sw_ScanElem()` to export all the other elements contained by `DOC` in whatever order you choose.

Note that after you use `Sw_ScanElem()`, the export event will then begin to process the parent element normally. In the above example, after exporting the elements contained by `DOC`, FrameMaker will proceed to export element contents of `DOC`. The surprising result is that all the elements contained by `DOC` will be exported again, in their natural order. To avoid this, you must drop the parent element contents after using `Sw_ScanElem()` to reorder its child elements.

The use of this function assumes your knowledge of the elements to be contained by the parent element, as well as the expected order of them in the exported markup document. Also note that the `Id` of the element to scan is not contained in the current conversion object. To get `elemId`, use standard FDK functions to access the list of elements that are children of the element associated with the current conversion object.

`Sw_ScanElem()` provides the means to reorder child elements when exporting from FrameMaker to a markup document. It does not alter the FrameMaker document in any way. For round-trip conversions, you might need to reorder the elements of an imported markup document. To do this, use `Sr_SetInsertLoc()` to determine the order in which a group of child elements will be inserted into their parent element.

Returns

On error, one of the following error codes:

- `SRW_E_NOT_CUR_DOC_ID`
- `SRW_E_BAD_OBJ_HANDLE`
- `SRW_E_FAILURE`

Examples

The following code checks for a `first` element contained in a `parent` element. If it finds one, it exports `first` as the first child of `parent`. Then it loops through the contents of `parent`, exporting all elements in order except for `first`, which the loop skips over.

```
#include "fm_struct.h"
#include "fstrings.h"

/* The element whose content will be re-ordered. */
#define PARENT ((StringT)"body")
/* The child that comes first - in markup */
#define FIRST ((StringT)"first")

SrwErrorT Sw_EventHandler(eventp, swObj)
SwEventT *eventp;
SwConvObjT swObj;
{
    SwConvObjT docObj;
    F_ObjHandleT docId, tmpId, elemId, defId;
    StringT elemGi;
    IntT flags;

    if ((eventp->evtype) == SW_EVT_BEGIN_ELEM) {
        docObj = Sw_GetCurConvObjOfType(SW_OBJ_DOC);
        docId = Sw_GetDocId(docObj);
        elemGi = Sw_GetStructuredGi(swObj);
        elemId = eventp->fm_elemid;

        /* If you have PARENT, convert it, then scan contents. */
        if (F_StrIEqual(elemGi, PARENT)) {
            F_ApiDeallocateString(&elemGi);
            flags = SRW_DROP_ELEMENT_CONTENT;
            Sw_SetProcessingFlags(swObj, flags);
            Sw_Convert(eventp, swObj);
        }
    }
}
```

```
/* Loop for FIRST and convert it using Sw_ScanElem(). */
    tmpId = F_ApiGetId(docId, elemId, FP_FirstChildElement);
    while (tmpId) {
        defId = F_ApiGetId(docId, tmpId, FP_ElementDef);
        elemGi = F_ApiGetString(docId, defId, FP_Name);
        if (F_StrIEqual(elemGi, FIRST))
            Sw_ScanElem(docId, tmpId);

        if(!F_StrIsEmpty(elemGi))
            F_ApiDeallocateString(&elemGi);
        tmpId = F_ApiGetId(docId, tmpId,
                            FP_NextSiblingElement);
    }

/* Now export the rest of PARENT, but skip FIRST */
    tmpId = F_ApiGetId(docId, elemId, FP_FirstChildElement);
    while (tmpId) {
        defId = F_ApiGetId(docId, tmpId, FP_ElementDef);
        elemGi = F_ApiGetString(docId, defId, FP_Name);
        if (!(F_StrIEqual(elemGi, FIRST)))
            Sw_ScanElem(docId, tmpId);

        if(!F_StrIsEmpty(elemGi))
            F_ApiDeallocateString(&elemGi);
        tmpId = F_ApiGetId(docId, tmpId,
                            FP_NextSiblingElement);
    }

/* Return SRW_E_SUCCESS... PARENT has already been converted */
    return(SRW_E_SUCCESS);
} /*End of the IF PARENT clause*/
if(!F_StrIsEmpty(elemGi))
    F_ApiDeallocateString(&elemGi);
} /*End of the IF SW_EVT_BEGIN_ELEM clause*/
return(Sw_Convert(eventp, swObj));
}
```

The following example reverses the effect of using `Sw_ScanElem()` to reorder elements; it imports the markup document and reverses the order of the elements in question. The code checks for a *first* element, which should be the first child element contained in a parent

element. If it finds one, it imports `first` as the last child in `parent`. Then it imports every other child of `parent` in order, placing them ahead of `first` in the list of sibling elements.

```
#include "fm_struct.h"
#include "fstrings.h"

/* The element whose content will be re-ordered. */
#define PARENT ((StringT)"body")
/* The child that comes first - in markup*/
#define FIRST ((StringT)"first")

SrwErrorT Sr_EventHandler(eventp, srObj)
SrEventT *eventp;
SrConvObjT srObj;
{
    static F_ObjHandleT parentElemId, firstElemId = 0;
    static BoolT haveParent = False;
    SrInsertLocT insertLocation;

    switch(eventp->evtype) {
    case SR_EVT_END_ELEM:
/* If finished with parent, set static vars.*/
        if (F_StrIEqual(eventp->u.tag.gi, PARENT)) {
            haveParent = False;
            firstElemId = 0;
        }
        break;

    case SR_EVT_BEGIN_ELEM:
/* If pPARENT, create the element in the FrameMaker */
/* document. This establishes the empty container element. */
/* Then save the ID of that container element. */
        if (F_StrIEqual(eventp->u.tag.gi, PARENT)) {
            Sr_Convert(eventp, srObj);
            parentElemId = Sr_GetFmElemId(srObj);
            haveParent = True;
            return(SRW_E_SUCCESS);
        }

/* If this is not a parent element, check whether */
/* it's a child of PARENT. */
        if (haveParent) {

/* If this is FIRST, insert as the last child of PARENT */
/* and store elem ID in firstElemId. */
```

```
    if (F_StrIEqual(eventp->u.tag.gi, FIRST)) {
        insertLocation.pos = SR_LOC_ELEMENT;
        insertLocation.u.elemLoc.parentId = parentElemId;
        insertLocation.u.elemLoc.childId = 0;
        insertLocation.u.elemLoc.offset = 0;
        Sr_SetInsertLoc(srObj, &insertLocation);
        Sr_Convert(eventp, srObj);
        firstElemId = Sr_GetFmElemId(srObj);
        return(SRW_E_SUCCESS);
    } else { /* Child of PARENT, but not FIRST */
/* If you have a firstElemId, set the insert location */
/* just previous to it. In either case, then convert */
/* the element normally. */
        if (firstElemId) {
            insertLocation.pos = SR_LOC_ELEMENT;
            insertLocation.u.elemLoc.parentId = parentElemId;
            insertLocation.u.elemLoc.childId = firstElemId;
            insertLocation.u.elemLoc.offset = 0;
            Sr_SetInsertLoc(srObj, &insertLocation);
        }
        return(Sr_Convert(eventp, srObj));
    } /* End of IF FIRST */
} /* End of IF haveParent */
break;
default:
    break;
} /* End of SWITCH */
/* If no above conditions were true, we still */
/* need to convert the event. */
return(Sr_Convert(eventp, srObj));
}
```

See also

- [“Sr_SetInsertLoc\(\)” on page 210](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on F_ObjHandleT

Sw_SetAttrVal()

Sets the markup attribute value for a passed attribute associated with the current export conversion object.

Synopsis

```
#include "fm_struct.h"

. . .
SrwErrorT Sw_SetAttrVal(SwConvObjT swObj, StructuredAttrValT
*attVal);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>attVal</i>	Pointer to the attribute-value structure to use

Details

This function sets the passed attribute value structure to the current conversion object. Setting an attribute value is only valid if the specified conversion object is associated with an event of type `SW_EVT_BEGIN_ELEM.`

You can use this function to set an attribute value to an element in markup, even if the exported FrameMaker element has no attributes or attribute values. The attribute name and value you provide must be valid for the exported document's doc type. For more information about how FrameMaker translates attributes, see *FrameMaker Structure Application Developer's Guide*.

This function is passed a structure of type `StructuredAttrValT`, which is defined as:

```
typedef struct (
    StringT sgmlAttrName;
    StringListT sgmlAttrVal;
    IntT sgmlAttrFlags;
} StructuredAttrValT;
```

Note that `sgmlAttrVal` is a string list. Attribute values in markup can be character data, or they can be tokens or lists of tokens. If the value corresponds to character data or a single token, then you should set only one element in the list of strings. If the value corresponds to a list of tokens, then each string in the list corresponds to a single value token.

To set an attribute value, you must call this function before the object is converted.

Important: When you no longer need the attribute-value data structure, call `Structured_DeallocateAttrVal()` to delete the structure and free the memory allocated for it.

To get an attribute value associated with a conversion object, call `Sr_GetAttrVal()`.

Returns

On error, one of the following error codes:

- `SRW_E_BAD_VALUE`
- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_NO_SUCH_ATTR`

Examples

The following code retrieves the attribute-value data structure for a `LIST_TYPE` attribute associated with the current conversion object. It then checks for the element type of the list's parent; if the parent is a checklist, the code changes the attribute for bullet type from a bullet to a check box.

Note that the attribute value in markup is a list of name tokens. To add a token to the list, this code appends a value to the string list, `curAttr.values`. After changing the attribute value structure, it sets the structure to the current conversion object.

```
. . .
SrwErrorT Sw_EventHandler(eventp, swObj)
SwEventT *eventp;
SwConvObjT swObj;
{
    StructuredAttrValT curAttr;
    ConStringT curVal, curGi;
    SwConvObjT parentObj;

    if (eventp->evtype == SW_EVT_BEGIN_ELEM) {
        if (F_StrIEqual(Sw_GetStructuredGi(swObj),
            (StringT) "LIST")) {
            parentObj = Sw_GetParentConvObj(swObj);
            curAttr = Sw_GetAttrVal(swObj, (StringT) "LIST_TYPE");
            /* no need to free strings from F_StrListGet */
            curVal = F_StrListGet(curAttr.sgmlAttrVal, 0);
            curGi = Sw_GetStructuredGi(parentObj);

            if (F_StrIEqual(curVal, (StringT) "BUL") &&
                F_StrIEqual(curGi, (StringT) "CHCKLST"))
            {
                F_StrListSetString(curAttr.sgmlAttrVal,
                    (StringT) "box", (UIntT) 0);
            }
        }
    }
}
```

```
        Sw_SetAttrVal(swObj, &curAttr);
        Structured_DeallocateAttrVal(&curAttr);
        F_ApiDeallocateString(&curGi);
    }
}
return (Sw_Convert(eventp, swObj));
}
```

See also

- [“Sw_GetAttrVal\(\)” on page 290](#)
- [“Sw_SetAttrVals\(\)” on page 347](#)
- [“SwConvObjT” on page 424](#)
- [“StructuredAttrValT” on page 412](#)
- [“SrwErrorT” on page 396](#)

Sw_SetAttrVals()

Specifies the list of attribute values to use with the current export conversion object.

Synopsis

```
#include "fm_struct.h"

. . .
SrwErrorT *Sw_SetAttrVals(SwConvObjT swObj,
    StructuredAttrValsT *attVals);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>attVals</i>	Pointer to the attribute value list to use

Details

This function passes the list of attribute values to use with the current conversion object. The object can only have attribute values if the current conversion object is associated with an event of type `SR_EVT_BEGIN_ELEM`.

You can use this function to set a list of attribute values to an element in markup, even if the exported FrameMaker element has no attributes or attribute values. The attribute names and values you provide must be valid for the exported document's doc type. For more information about how FrameMaker translates attributes, see *FrameMaker Structure Application Developer's Guide*.

This function is passed a structure of type `StructuredAttrValsT`, which is defined as:

```
typedef struct {
    UIntT len; /* number of attribute values */
    StructuredAttrValT *val; /* attribute values array pointer */
} StructuredAttrValsT;
```

`val` points to an array of attribute values, each one represented by a structure of type `StructuredAttrValT`, which is defined as:

```
typedef struct (
    StringT sgmlAttrName;
    StringListT sgmlAttrVal;
    IntT sgmlAttrFlags;
} StructuredAttrValT;
```

Note that `sgmlAttrVal` is a list of strings. Attribute values in markup can be character data, or they can be tokens or lists of tokens. If the value corresponds to character data or a single token, then you should set only one element in the list of strings. If the value corresponds to a list of tokens, then each string in the list corresponds to a single value token.

For more information about single attribute values, see [“Sw_SetAttrVal\(\)” on page 345](#).

To set a list of attribute values, you must call this function before the object is converted.

Important: When you no longer need the `F_AttributesT` data structure, call `Structured_DeallocateAttrVals()` to delete the structure and free the memory allocated for it.

Returns

On error, one of the following error codes:

- `SRW_E_BAD_VALUE`
- `SRW_E_WRONG_OBJ_TYPE`

Examples

The following code retrieves the list of attribute values associated with the current conversion object. It then checks the list's parent to see if the list is part of a `CHECK_LIST` element. If it is, it changes every occurrence of an attribute value token of `BULLET` to `CHECK_MARK`. To do that, the code uses two nested loops, first to find the `LIST_TYPE` attribute, then to find any tokens of `BULLET` in the `LIST_TYPE` attribute. After changing the attribute values structure, it sets the structure to the current conversion object.

```
. . .
SrwErrorT Sw_EventHandler(eventp, swObj)
SwEventT *eventp;
SwConvObjT swObj;
```

```
{
    StructuredAttrValsT curAttrVals;
    StringT s;
    SwConvObjT parentObj;
    UIntT i;

    if (eventp->evtype == SW_EVT_BEGIN_ELEM &&
        F_StrIEqual(Sw_GetStructuredGi(swObj), (StringT)"LIST"))
    {
        parentObj = Sw_GetParentConvObj(swObj);
        s = Sw_GetStructuredGi(parentObj);
        if(F_StrIEqual(s, (StringT) "CHCKLST")) {
            curAttrVals = Sw_GetAttrVals(swObj);
            for (i=0;i<curAttrVals.len;i++) {
                if (F_StrIEqual(curAttrVals.val[i].sgmlAttrName,
                                "LIST_TYPE")) {
                    F_StrListSetString(curAttrVals.val[i].sgmlAttrVal,
                                        (StringT)"box", (UIntT)0);
                    break;
                }
            }
            Sw_SetAttrVals(swObj, &curAttrVals);
            Structured_DeallocateAttrVals(&curAttrVals);
        }
        F_ApiDeallocateString(&s);
    }
    return (Sw_Convert(eventp, swObj));
}
```

See also

- [“Sw_GetAttrVals\(\)” on page 293](#)
- [“Sw_SetAttrVal\(\)” on page 345](#)
- [“SwConvObjT” on page 424](#)
- [“StructuredAttrValsT” on page 413](#)
- [“SrwErrorT” on page 396](#)

Sw_SetBookCompEntityFilePath()

Specifies the filepath to use for exporting an export conversion object of type `SW_OBJ_BOOK_COMP`.

Synopsis

```
#include "fm_struct.h"

. . .
SrwErrorT Sw_SetBookCompEntityFilePath(SwConvObjT swObj,
FilePathT *filePath);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>filePath</i>	Pointer to the filepath to use

Details

To specify the filepath to use for exporting a FrameMaker book component (a document that is part of a book), call `Sw_SetBookCompEntityFilePath()` before converting the object. *swObj* must be of type `SW_OBJ_BOOK_COMP`.

`FilePathT` is a pointer to a platform-independent representation of a platform-specific path. For more information about working with platform-independent representations of filepaths, see Chapter 16, "Making I/O and Memory Calls Portable," in the *FDK Programmer's Guide*.

Returns

On error, one of the following error codes:

- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_NOT_BOOK_COMP`

Examples

The following code sets the filepath to use for exporting a FrameMaker book component:

```
. . .
SrwErrorT errorStatus;
FilePathT *bookCompFp;

. . .
/* The following call assumes that bookCompFp is initialized. */
if(Sw_GetObjType(swObj) == SW_OBJ_BOOK_COMP)
    errorStatus = Sw_SetBookCompEntityFilePath(swObj,
        bookCompFp);
. . .
```

See also

- [“Sw_GetBookCompEntityFilePath\(\)” on page 295](#)
- [“SwConvObjT” on page 424](#)
- [“SrwErrorT” on page 396](#)
- [“FilePathT” on page 409](#)

Sw_SetBookCompPi()

Sets the *processing instructions (PI)* for a FrameMaker book component associated with the specified export conversion object of type `SW_OBJ_BOOK_COMP`.

Synopsis

```
#include "fm_struct.h"

. . .

SrwErrorT Sw_SetBookCompPi(SwConvObjT swObj, StringT bookCompPi);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>bookCompPi</i>	<i>Processing instructions</i> to use

Details

To specify the *PI* for an export conversion object associated with a book component (a document that is part of a FrameMaker book file), call `Sw_SetBookCompPi()` before converting the object. *swObj* must be of type `SW_OBJ_BOOK_COMP`.

To specify the *PI* for an entire book, call `Sw_SetBookPi()` instead.

bookCompPi is a string that should not exceed the *processing instruction length (PILEN)* quantity specified in the SGML declaration for the markup document. This function does not check the length of *bookCompPi* against the *PILEN* quantity, nor does it check to see if the *PI* close (*PIC*) delimiter occurs within *bookCompPi*. If the *PIC* delimiter occurs within *bookCompPi*, during processing *bookCompPi* is truncated when the *PIC* is encountered.

If *bookCompPi* is `NULL`, `Sw_Convert()` does not generate a *PI*.

Returns

On error, one of the following error codes:

- `SRW_E_NOT_BOOK_COMP`
- `SRW_E_WRONG_OBJ_TYPE`

Examples

The following code sets the *PI* for a FrameMaker book component represented by the current conversion object:

```
. . .
SrwrErrorT errorStatus;
StringT processingInstruction;
. . .
if (Sw_GetObjType(swObj) == SW_OBJ_BOOK_COMP)
    errorStatus = Sw_SetBookCompPi(swObj,
        processingInstruction);
. . .
```

See also

- [“Sw_GetBookCompPi\(\)” on page 296](#)
- [“Sw_SetBookPi\(\)” on page 352](#)
- [“SwConvObjT” on page 424](#)
- [“SrwrErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on StringT

Sw_SetBookPi()

Sets the *processing instructions (PI)* for a FrameMaker book associated with the specified export conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
SrwrErrorT Sw_SetBookPi(SwConvObjT swObj, StringT bookPi);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>bookPi</i>	<i>Processing instructions</i> to use

Details

To set the *processing instructions (PI)* for a FrameMaker book associated with an export conversion object, call `Sw_SetBookPi()` before converting the object. *swObj* must be of type `SW_OBJ_BOOK`.

bookPi is a string that should not exceed the *processing instruction length (PILEN)* quantity specified in the SGML declaration for the markup document. This function does not check the length of *bookPi* against the *PILEN* quantity, nor does it check to see if the *PI close*

(*PIC*) delimiter occurs within *bookPi*. If the *PIC* delimiter occurs within *bookPi*, during processing *bookPi* is truncated when the *PIC* is encountered.

If *bookPi* is NULL, `Sw_Convert()` does not generate a *PI*.

To set a *PI* for a book component (a document that is part of a FrameMaker book file), call `Sw_SetBookCompPi()`.

Returns

On error, `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code sets the *PI* for a FrameMaker book represented by the current conversion object:

```
. . .
SrwErrorT errorStatus;
StringT piStr;
. . .
if(Sw_GetObjType(swObj) == SW_OBJ_BOOK)
    errorStatus = Sw_SetBookPi(swObj, piStr);
. . .
F_ApiDeallocateString(&piStr);
. . .
```

See also

- [“Sw_GetBookPi\(\)” on page 297](#)
- [“Sw_SetBookCompPi\(\)” on page 351](#)
- [“SwConvObjT” on page 424](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_SetColSpecs()

Specifies a list of *CALS* column specifications (*colspecs*) to associate with a specified table export conversion object of type `SW_OBJ_TABLE`.

Synopsis

```
#include "fm_struct.h"

. . .
SrwErrorT Sw_SetColSpecs(SwConvObjT swObj,
    SrwColSpecsT *listp);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>listp</i>	<i>Colspecs</i> list to assign

Details

To specify the list of *colspecs* to use for a table associated with an export conversion object of type `SW_OBJ_TABLE`, call `Sw_SetColSpecs()` before converting the object. This function should be used only to export a *CALS* table.

Important: You cannot specify a *colspec* list for a table part (table heading, table body, table footing).

When you no longer need a list of *colspecs*, delete it and free the memory allocated for it with `Srw_DeallocateColSpecs()`.

Returns

On error, `SRW_W_WRONG_OBJ_TYPE`.

Examples

The following code, from an event handler for a *CALS* table object, retrieves a *colspec* list, modifies it, and then assigns the modified list to the table object:

```
. . .
SrwErrorT Sw_EventHandler(SwEventT *eventp, SwConvObjT swObj)
{
    SrwColSpecsT colspecs;
    SrwColSpecT colspec;
    SwConvObjT tblObj;
```

```
switch(Sw_GetObjType(swObj))
{
    case SW_OBJ_TABLE_ROW:
        /* Retrieve current table object. */
        tblObj = Sw_GetCurConvObjOfType(SW_OBJ_TABLE);
        /* Retrieve copy of colspecs for table object. */
        colspecs = Sw_GetColSpecs(tblObj);

        . . .

        /* Retrieve copy of colspec for the first column. */
        colspec = Srw_GetColSpecByColNum(&colspecs, 0);
        if(SRW_errno == SRW_E_SUCCESS)
        {
            /* Set the column width to 12 cm. */
            F_Free(colspec.width);
            colspec.width = F_StrCopyString("12cm");
            colspec.valueSet |= SRW_COLSPEC_COLWIDTH;
            /* Put new colspec back into list. */
            Srw_SetColSpec(&colspecs, &colspec);
            /* Put the new list back into the object. */
            Sw_SetColSpecs(tblObj, &colspecs);

            . . .

            /* Deallocate colspecs when done. */
            Srw_DeallocateColSpecs(&colspecs);
            Srw_DeallocateColSpec(&colspec);
        }

        . . .
}
```

See also

- [“Sw_GetColSpecs\(\)” on page 300](#)
- [“Srw_DeallocateColSpecs\(\)” on page 251](#)
- [“SwConvObjT” on page 424](#)
- [“SrwColSpecsT” on page 419](#)
- [“SrwErrorT” on page 396](#)

Sw_SetEntityName()

Sets the name for an entity in markup associated with an export conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

SrwErrorT Sw_SetEntityName(SwConvObjT swObj,
StringT entName);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>entName</i>	Name of the entity to use

Details

To specify the name for an entity in markup associated with an export conversion object, call `Sw_SetEntityName()` before converting the object.

To retrieve the current entity name for a conversion object, call `Sw_GetEntityName()`.

Returns

On error, `SRW_E_WRONG_OBJ_TYPE` .

Examples

The following code sets the entity name for the current conversion object:

```
. . .
StringT entName;
SrwErrorT errorStatus;
. . .
errorStatus = Sw_SetEntityName(swObj, entName);
. . .
```

See also

- [“Sw_GetEntityName\(\)” on page 306](#)
- [“SwConvObjT” on page 424](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_SetExportFileFormat()

Specifies the file format to use when exporting a graphic or equation associated with an export conversion object of type SW_OBJ_GRAPHIC or SW_OBJ_EQUATION.

Synopsis

```
#include "fm_struct.h"

. . .
SrwErrorT Sw_SetExportFileFormat(SwConvObjT swObj,
StringT format);
```

Arguments

- swObj* Conversion object to change
- format* String containing file format and filter to use

Details

To set the file format to use when exporting a graphic or equation associated with an SW_OBJ_GRAPHIC or SW_OBJ_EQUATION export conversion object, call Sw_SetExportFileFormat() before converting the object.

Depending on the file format you specify, an export filter may be necessary to convert the object. The following table describes how FrameMaker processes a graphic or equation conversion object based on the file format information, and the number of objects in the anchored frame:

File Format	Number of objects	Strategy
NULL	1	FrameMaker examines the object's inset facet for a file format other than FrameVector or FramedImage. If a recognizable file format is found, the object is exported as is, without invoking a filter. Otherwise, FrameMaker invokes the CGM filter to export the object.
NULL	> 1	FrameMaker invokes the CGM filter to export the entire anchored frame.
non-NULL	1	FrameMaker examines the object's inset facet for a matching file format. If a matching format is found, the object is exported as is, without invoking a filter. Otherwise, FrameMaker attempts to convert the object by invoking a filter for the specified file format.
non-NULL	> 1	FrameMaker attempts to convert the entire anchored frame by invoking a filter for the specified file format.

If you specify a file format with Sw_SetExportFileFormat(), choose a format for which you have the necessary filter.

format must be a string corresponding to the file format to use. For example, CGM is a valid file format. See the *FDK Platform Guide* for your platform and the release notes for export filter information.

To determine the current file format for export, call `Sw_GetExportFileFormat()`.

To set the filepath to be used for exporting, call `Sw_SetExportFilePath()`.

Returns

On error, one of the following error codes:

- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_BAD_VALUE`

Examples

The following code sets the export file format for a conversion object of type `SW_OBJ_GRAPHIC`:

```
. . .
    SrwErrorT errorStatus = SRW_E_SUCCESS;
    . . .
    if(Sw_GetObjType(swObj) == SW_OBJ_GRAPHIC)
        errorStatus = Sw_SetExportFileFormat(swObj, (StringT)"CGM");
    . . .
```

See also

- [“Sw_GetExportFileFormat\(\)” on page 307](#)
- [“Sw_SetExportFilePath\(\)” on page 359](#)
- [“SwConvObjT” on page 424](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_SetExportFilePath()

Specifies the platform-independent filepath to use to export a graphic or equation associated with an export conversion object.

Synopsis

```
#include "fm_struct.h"

. . .
SrwErrorT Sw_SetExportFilePath(SwConvObjT swObj,
    FilePathT *filePath);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>filePath</i>	Pointer to filepath to use

Details

To set the export filepath for a graphic or equation associated with an `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION` export conversion object, call `Sw_SetExportFilePath()` **before** converting the object.

If *filePath* is valid, then on export to markup, FrameMaker creates a separate file containing the exported graphic or equation.

To export a graphic or equation to the s document instance without creating a separate graphic or equation, set *filePath* to `NULL`.

`FilePathT` is a pointer to a platform-independent representation of a platform-specific path. For more information about working with platform-independent representations of filepaths, see Chapter 16, "Making I/O and Memory Calls Portable," in the *FDK Programmer's Guide*.

Returns

On error, one of the following error codes:

- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_BAD_VALUE`

Examples

The following code sets the export file path for a graphic associated with the current conversion object if it is of type `SW_OBJ_GRAPHIC`:

```
. . .
SrwErrorT errorStatus = SRW_E_SUCCESS;
FilePathT *fileLocation;
. . .
```

Sw_SetGfxDataAttrVals()

```
if(Sw_GetObjType(swObj) == SW_OBJ_GRAPHIC)
    errorStatus = Sw_SetExportFilePath(swObj, fileLocation);
. . .
```

See also

- [“Sw_ExportFilePath\(\)” on page 309](#)
- [“Sw_SetExportFileFormat\(\)” on page 357](#)
- [“SwConvObjT” on page 424](#)
- [“FilePathT” on page 409](#)
- [“SrwErrorT” on page 396](#)

Sw_SetGfxDataAttrVals()

Specifies the list of data attributes to use for a graphic or equation associated with an export conversion object of type `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION`.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sw_SetGfxDataAttrVals(SwConvObjT swObj
    SgmlAttrVals *attrVals);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>attrVals</i>	Pointer to list of attributes to use

Details

To assign a list of data attributes to a graphic or equation associated with an `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION` export conversion object, call `Sw_SetGfxDataAttrVals()` before converting the object.

Important: When you no longer need the attribute list structure used by this function, free it with a call to `Structured_DeallocateAttrVals()`.

To retrieve a list of data attributes for a graphic or equation export conversion object, call `Sw_GetGfxDataAttrVals()`.

Returns

On error, `SRW_E_WRONG_OBJ_TYPE`.

Examples

For more detailed examples of handling attribute values, see [“Sw_GetAttrVal\(\)” on page 290](#), [“Sw_GetAttrVals\(\)” on page 293](#), [“Sw_SetAttrVal\(\)” on page 345](#), and [“Sw_SetAttrVals\(\)” on page 347](#). The following code sets the attribute list for graphic or equation associated with the current conversion object of type `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION`:

```
. . .
SrwErrorT errorStatus;
StructuredAttrValsT attributes;
. . .
if((Sw_GetObjType(swObj) == SW_OBJ_GRAPHIC) ||
    (Sw_GetObjType(swObj) == SW_OBJ_EQUATION))
    errorStatus = Sw_SetGfxDataAttrVals(swObj, &attributes);
. . .
```

See also

- [“Sw_GetGfxDataAttrVals\(\)” on page 310](#)
- [“SwConvObjT” on page 424](#)
- [“StructuredAttrValsT” on page 413](#)
- [“SrwErrorT” on page 396](#)

Sw_SetGfxEntityName()

Sets the entity name for the graphic or equation associated with the specified export conversion object of type `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION`.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sw_SetGfxEntityName(SwConvObjT swObj, StringT entName);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>entName</i>	Entity name to assign

Details

To set or change the entity name for a graphic or equation associated with a specific export conversion object of type `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION`, call `Sw_SetGfxEntityName()` before converting the object.

To determine the entity name for a graphic or equation, call `Sw_GetGfxEntityName()`.
To set a graphic entity's type, call `Sw_SetGfxEntityType()`.

Returns

On error, `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code sets the entity name for the graphic or equation associated with the current conversion object if it is of type `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION`:

```
. . .
SrErrorT errorStatus;
StringT entityName;
. . .
if((Sw_GetObjType(swObj) == SW_OBJ_GRAPHIC) ||
    (Sw_GetObjType(swObj) == SW_OBJ_EQUATION))
    errorStatus = Sw_SetGfxEntityName(swObj, entityName);
. . .
```

See also

- [“Sw_GetGfxEntityName\(\)” on page 311](#)
- [“Sw_SetGfxEntityType\(\)” on page 362](#)
- [“SwConvObjT” on page 424](#)
- [“SrErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_SetGfxEntityType()

Sets or changes the entity type for the graphic or equation associated with the specified export conversion object of type `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION`.

Synopsis

```
#include "fm_struct.h"
. . .
SrErrorT Sw_SetGfxEntityType(SwConvObjT swObj,
    structuredEntityTypeT entType));
```

Arguments

<i>swObj</i>	Conversion object to change
<i>entType</i>	Entity type to set

Details

Call `Sw_SetGfxEntityType()` to set or change the entity type to assign for a graphic or equation associated with an `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION` export conversion object before converting the object.

entType should be set to one of the following values:

- `STRUCTURED_ET_CDATA`
- `STRUCTURED_ET_SDATA`
- `STRUCTURED_ET_NDATA`

To determine the type of entity assigned to a graphic or equation export conversion object, call `Sw_GetGfxEntityType()`. To set the name of the entity associated with a graphic or equation export conversion object, call `Sw_SetGfxEntityName()`.

Returns

On error, one of the following error codes:

- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_BAD_VALUE`

Examples

The following code sets the entity type for a graphics export conversion object to *CDATA*:

```
. . .
SrwrErrorT errorStatus;
. . .
if(Sw_GetObjType(swObj) == SW_OBJ_GRAPHIC)
    errorStatus = Sw_SetGfxEntityType(swObj, STRUCTURED_ET_CDATA);
. . .
```

See also

- [“Sw_GetGfxEntityType\(\)” on page 312](#)
- [“Sw_SetGfxEntityName\(\)” on page 361](#)
- [“SwConvObjT” on page 424](#)
- [“StructuredEntityTypeT” on page 389](#)
- [“SrwrErrorT” on page 396](#)

Sw_SetGfxNotation()

Sets the *notation name* for the specified export conversion object of type `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION`.

Synopsis

```
#include "fm_struct.h"

. . .

SrwErrorT Sw_SetGfxNotation(SwConvObjT swObj, StringT notName);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>notName</i>	<i>Notation name</i> to assign

Details

To set the *notation name* for a graphic or equation associated with an `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION` export conversion object, call `Sw_SetGfxNotation()` **before** converting the object.

To determine the current *notation name* for a graphic or equation conversion object, call `Sw_GetGfxNotation()`.

Returns

On error, `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code sets the *notation name* for an equation associated with the current conversion object:

```
. . .
SrwErrorT errorStatus
StringT notationName;
. . .
if(Sw_GetObjType(swObj) == SW_OBJ_EQUATION)
    errorStatus = Sw_SetGfxNotation(swObj, notationName);
. . .
```

See also

- [“Sw_GetGfxNotation\(\)” on page 314](#)
- [“SwConvObjT” on page 424](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_SetGfxPubId()

Sets or changes the *public identifier* for the specified export conversion object of type SW_OBJ_GRAPHIC or SW_OBJ_EQUATION.

Synopsis

```
#include "fm_struct.h"

. . .

SrwErrorT Sw_SetGfxPubId(SwConvObjT swObj, StringT pubId);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>pubId</i>	String containing <i>public identifier</i> to use

Details

To set or change the *public identifier* for a graphic or equation associated with a specific export conversion object of type SW_OBJ_GRAPHIC or SW_OBJ_EQUATION, call Sw_SetGfxPubId() before converting the object. This function does not check that the *pubId* conforms to ISO standards for *public identifiers*.

To determine the *public identifier* for a graphic or equation export conversion object, call Sw_GetGfxPubId(). To set the *system identifier* for a graphic or equation export conversion object, call Sw_SetGfxSysId().

Returns

On error, SRW_E_WRONG_OBJ_TYPE .

Examples

The following code sets the *public identifier* for the graphic or equation associated with the current conversion object:

```
. . .
SrwErrorT errorStatus;
StringT publicIdentifier;
. . .
if((Sw_GetObjType(swObj) == SW_OBJ_GRAPHIC) ||
    (Sw_GetObjType(swObj) == SW_OBJ_EQUATION))
    errorStatus = Sw_SetGfxPubId(swObj, publicIdentifier);
. . .
```

See also

- [“Sw_GetGfxPubId\(\)” on page 315](#)
- [“Sw_SetGfxSysId\(\)” on page 366](#)
- [“SwConvObjT” on page 424](#)

- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_SetGfxSysId()

Sets or changes the *system identifier* for the specified export conversion object of type `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION`.

Synopsis

```
#include "fm_struct.h"

. . .

SrwErrorT Sw_SetGfxSysId(SwConvObjT swObj, StringT sysId);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>sysId</i>	String containing <i>system identifier</i> to use

Details

Call `Sw_SetGfxSysId()` to set or change the *system identifier* for a graphic or equation associated with a specific export conversion object of type `SW_OBJ_GRAPHIC` or `SW_OBJ_EQUATION` before converting the object.

To determine the *system identifier* for a graphic or equation export conversion object, call `Sw_GetGfxSysId()`. To set the *public identifier* for a graphic or equation export conversion object, call `Sw_SetGfxPubId()`.

Returns

On error, `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code sets the *system identifier* for a graphic associated with the current conversion object:

```
. . .
SrwErrorT errorStatus;
StringT systemIdentifier;
. . .
if(Sw_GetObjType(swObj) == SW_OBJ_GRAPHIC)
    errorStatus = Sw_SetGfxSysId(swObj, systemIdentifier);
. . .
```

See also

- [“Sw_GetGfxSysId\(\)” on page 316](#)
- [“Sw_SetGfxPubId\(\)” on page 365](#)

- [“SwConvObjT” on page 424](#)
- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_SetPI()

Sets the *processing instructions (PI)* associated with an export conversion object of type `SW_OBJ_PI`.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sw_SetPI(SwConvObjT swObj, StringT piStr);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>piStr</i>	<i>PI</i> to set

Details

To set the *PI* for the specified export conversion object of type `SW_OBJ_PI` before it is converted, call `Sw_SetPI()`. This function does not verify that the *PI* you specify is valid.

To retrieve the current *PI* for a conversion object, call `Sw_GetPI()`.

Returns

On error, `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code sets the *PI* for the current conversion object of type `SW_OBJ_PI`:

```
. . .
SwConvObjT piObj;
StringT piStr;
SrwErrorT errorStatus;
. . .
piObj = Sw_GetCurConvObjOfType(SW_OBJ_PI);
errorStatus = Sw_SetPI(piObj, piStr);
. . .
```

See also

- [“Sw_GetPI\(\)” on page 319](#)
- [“SwConvObjT” on page 424](#)
- [“SrwErrorT” on page 396](#)

- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_SetPrivateData()

Sets to the specified conversion object a pointer to the private data your application allocates, initializes, and maintains.

Synopsis

```
#include "fm_struct.h"
. . .
VoidT Sw_SetPrivateData(SwConvObjT swObj, PtrT privDatap);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>privDatap</i>	Pointer to data to insert as private data

Details

You can allocate and maintain data that is private to your application, and then assign the address of that data to a specific conversion object. This is a way to save a state within your application, and then associate it with a specific level in the hierarchy of the conversion object stack. This function sets a pointer to such data to the specified conversion object.

Note that your application owns the data. Also, the data can be of any addressable type, from a single variable to a structure within a deep hierarchy of structures. It is up to your application to maintain the data content.

Important: When the conversion object is closed, it is removed from the conversion object stack and its allocated memory is freed. However, the data referenced by the private data pointer is not deallocated. If the only reference to this data is stored with the conversion object, and that object is closed, you will not be able to deallocate the private data. Be sure to trap ending events (for example, `SR_EVT_END_ELEM`) and check their associated objects for private data pointers.

Returns

`VoidT`.

Examples

For an example of setting private data to a conversion object, see [“Sw_GetPrivateData\(\)” on page 320](#).

See also

- [“Sw_GetPrivateData\(\)” on page 320](#)
- [“SwConvObjT” on page 424](#)
- [“Primitive data types” on page 383](#) for more information on `PtrT`

Sw_SetProcessingFlags()

Sets the flags that indicate the processing for the specified export conversion object.

Synopsis

```
#include "fm_struct.h"

. . .

SrwErrorT Sw_SetProcessingFlags(SwConvObjT swObj, IntT flags);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>flags</i>	Integer containing desired processing flags

Details

To specify the processing for a conversion object corresponding to a proposed element, you must call `Sw_SetProcessingFlags()` before converting the object. The flags argument can be set to one of the following values:

- `NULL`

This is equivalent to no read/write rules.

- `0`

This overrides any read/write rules, and performs no processing.

- `SRW_UNWRAP_ELEMENT`

This is equivalent to the `unwrap` read/write rule.

- `SRW_DROP_ELEMENT_CONTENT`

This is equivalent to the `drop content` read/write rule.

- `SRW_UNWRAP_ELEMENT` and `SRW_DROP_ELEMENT_CONTENT`

Specified via a logical OR (`SRW_UNWRAP_ELEMENT | SRW_DROP_ELEMENT_CONTENT`).

This is equivalent to both the `unwrap` and `drop content` read/write rules for a single element.

The conversion object must correspond to a FrameMaker element. If you try to set the processing flags for a non element conversion object, this function returns an error.

Note that if you drop the content of an element, none of the child elements will post an event.

Returns

On error, one of the following error codes:

- `SRW_E_INVALID_CONV_OBJ`
- `SRW_E_WRONG_OBJ_TYPE`

- SRW_E_BAD_OBJ_HANDLE

Examples

For an example of using Sw_SetProcessingFlags(), see [“Sw_GetAttrVal\(\)” on page 290](#).

See also

- [“Sw_GetPrivateData\(\)” on page 320](#)
- [“SwConvObjT” on page 424](#)
- [“SwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for more information on IntT

Sw_SetSessionProps()

Specifies export session properties for the current session (as specified by a conversion object of type SW_OBJ_SESSION).

Synopsis

```
#include "fm_struct.h"
. . .
SwErrorT Sw_SetSessionProps(SwConvObjT swObj,
    SwSessionPropsT *sessionProps);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>sessionProps</i>	Pointer to the structure containing session property settings

Details

To set the current session properties, you pass a SwSessionPropsT structure and the current session conversion object. To get the current session object, call Sw_GetCurConvObjOfType(SR_OBJ_SESSION).

SwSessionPropsT is defined as:

```
typedef struct {
    BoolT includeDtd;
    BoolT includeSgmlDecl;
    StringT doctypeSysId;
    stringT doctypePubId;
    BoolT overWriteFiles; /*True if batch can overwrite files of
                           same name in the target directory*/
} SwSessionPropsT;
```

To retrieve the current export session properties, call Sw_GetSessionProps().

Returns

On error, one of the following error codes:

- `SRW_E_BAD_OBJ_HANDLE`
- `SRW_E_INVALID_CONV_OBJ`

Examples

The following code ensures a DTD and an SGML Declaration will be included with the exported SGML file. It declares an `SwSessionPropT` and assigns values to it. The code then assigns those properties to the current export session, and if successful, it finally frees the session properties structure:

```
. . .
SwSessionPropT exportSessionProps;
SrwErrorT errorStatus = SRW_E_SUCCESS;
SwConvObjT sessionObj;

. . .
sessionObj = SwGetCurConvObjOfType(SW_OBJ_SESSION);
exportSessionProps = Sw_GetSessionProps(sessionObj);
exportSessionProps.includeDtd = True;
exportSessionProps.includeSgmlDecl = True;

errorStatus = Sw_SetSessionProps(sessionObj,
                                &exportSessionProps);

. . .
Sw_DeallocateSessionProps(&exportSessionProps);

. . .
```

See also

- [“Sw_GetSessionProps\(\)” on page 329](#)
- [“Sw_DeallocateSessionProps\(\)” on page 284](#)
- [“Sw_GetCurConvObjOfType\(\)” on page 303](#)
- [“SwConvObjT” on page 424](#)
- [“SrwErrorT” on page 396](#)

Sw_SetStructuredGi()

Deprecated: **Sw_SetSgmlGi()**

Sets the markup *generic identifier (GI)* associated with the specified export conversion object.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sw_SetStructuredGi(SwConvObjT swObj, StringT gi);
```

Arguments

swObj Conversion object to change

gi *Generic identifier* to use

Details

To set the markup *GI* for an export conversion object, call `Sw_SetStructuredGi()`. This function automatically creates a new attribute list based on the specified *GI*. No values are specified for attributes in the new list except under the following conditions:

- Identical attribute names are assigned their values from the original attribute list.
- Attributes with default values are assigned their default values (but the flag indicating if a value is written in the start-tag is turned off).

No check is made to determine if the new *GI* is mapped to the same FrameMaker export conversion object as the original *GI*.

Returns

On error, `SRW_E_WRONG_OBJ_TYPE`.

Examples

The following code sets the markup *GI* for the current conversion object:

```
. . .
SrwErrorT errorStatus;
StringT giText;
. . .
errorStatus = Sw_SetStructuredGi(swObj, giText);
. . .
```

See also

- [“Sw_GetStructuredGi\(\)” on page 330](#)
- [“SwConvObjT” on page 424](#)
- [“SrwErrorT” on page 396](#)

- [“Primitive data types” on page 383](#) for more information on `StringT`

Sw_SetStructuredText()

Deprecated: *Sw_SetSgmlText()*

Sets the FrameMaker text associated with a conversion object for export to a markup document.

Synopsis

```
#include "fm_struct.h"

. . .

SrwErrorT Sw_SetStructuredText(SwConvObjT swObj,
    SwTextItemsT *textItemsp);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>textItemsp</i>	Pointer to text structure to use

Details

To specify the text that an export conversion object of type `SW_OBJ_TEXT` should pass to a markup document, call `Sw_SetStructuredText()`.

textItemsp is a pointer to a previously populated text items structure containing the text to use. Text might be a string, a numeric character reference, a named character reference, or an entity reference.

To retrieve text already associated with an export conversion object, call `Sw_GetStructuredText()`.

Returns

On error, one of the following error codes:

- `SRW_E_INVALID_TEXT_ITEM_TYPE`
- `SRW_E_WRONG_OBJ_TYPE`

Examples

The following code traps all the markup text FrameMaker proposes, and changes it to the text that is in string `s`:

```
. . .

SrwErrorT Sw_EventHandler(eventp, swObj)
    SwEventT *eventp;
    SwConvObjT swObj;
```

Sw_SetTableScanOrder()

```
{
    SwTextItemsT ti, newTi;
    SwTextItemT *tiVal;

    newTi.len = 1;

    tiVal = F_Calloc(1, sizeof(SwTextItemT), DSE);
    tiVal->itemType = SW_TEXT_STRING;
    tiVal->u.text = F_StrCopyString((StringT)"This is a string");
    newTi.val = tiVal;

    ti = Sw_GetStructuredText(swObj);
    if(ti.len > 0) {
        Sw_DeallocateTextItems(&ti);
        Sw_SetStructuredText(swObj, &newTi);
    }
    Sw_DeallocateTextItems(&newTi);
    return (Sw_Convert(eventp, swObj));
}
```

See also

- [“Sw_GetStructuredText\(\)” on page 332](#)
- [“SwConvObjT” on page 424](#)
- [“SwTextItemsT” on page 426](#)
- [“SrwErrorT” on page 396](#)

Sw_SetTableScanOrder()

Specifies the order in which FrameMaker scans table parts.

Synopsis

```
#include "fm_struct.h"
. . .
SrwErrorT Sw_SetTableScanOrder(SwConvObjT swObj,
    SrwTablePartTypeT *scanOrderp);
```

Arguments

<i>swObj</i>	Conversion object to change
<i>scanOrderp</i>	Pointer to order array

Details

To specify the order in which FrameMaker scans table parts (table title, table heading, table body, and table footing), call `Sw_SetTableScanOrder()`.

scanOrderp is a pointer to a four-element integer array that specifies the order in which to examine table parts. The first array element specifies the first table part to scan, the second array element the second table part, and so on. Table parts not specified in *scanOrderp* are not scanned.

To specify scan order for a subset of table parts, set a subset of array elements to valid values (found in `SrwTablePartTypeT`). Unused elements should be set to 0.

Returns

On error, one of the following error codes:

- `SRW_E_WRONG_OBJ_TYPE`
- `SRW_E_BAD_VALUE`.

Examples

The following code sets the table-part scan order to examine only table headings, followed by table footings:

```
. . .
SrwErrorT errorStatus;
IntT scanOrder[4], i;
. . .
/* Initialize all elements in ScanOrder. */
for(i = 0; i < 4; i++)
{
    scanOrder[i] = 0;
}
/* Set ScanOrder to heading first. */
scanOrder[0] = SRW_E_TABLE_HEADING;
/* Add Footing, too. */
scanOrder[1] = SRW_E_TABLE_FOOTING;
errorStatus = Sw_SetTableScanOrder(swObj, &scanOrder);
. . .
```

See also

- [“SwConvObjT” on page 424](#)
- [“SrwTablePartTypeT” on page 403](#)
- [“SrwErrorT” on page 396](#)

Sw_WriteAttrSpec()

Writes an attribute specification to the DTD subset for a data attribute, or to the document instance for an attribute in a start-tag.

Synopsis

```
#include "fm_struct.h"

. . .

VoidT Sw_WriteAttrSpec(SwLocationT loc,
StringT attrName,
StringT attrVal);
```

Arguments

<i>loc</i>	SW_LOC_DTD_SUBSET to write to the DTD, or SW_LOC_INSTANCE to write to the markup document instance
<i>attrName</i>	Name of the attribute for which to assign a value
<i>attrVal</i>	Attribute value to write

Details

To write an attribute specification to the DTD subset for a data attribute, or to the markup document instance for an attribute in a start-tag, call `Sw_WriteAttrSpec()`. The *loc* argument specifies where to write (and therefore whether the attribute is a data attribute or an attribute in a start-tag).

Important: If you write directly to the DTD or document instance, you should not pass the event/conversion object pair to the default conversion routine. Instead, you notify FrameMaker that you wrote a value out, and return control to FrameMaker so it can pass the next event/conversion pair to your custom handler.

This function can be called only after an `SW_EVT_BEGIN_DOC` or `SW_EVT_BEGIN_BOOK_COMP` event has occurred and before the corresponding `SW_EVT_END_DOC` or `SW_EVT_END_BOOK_COMP` event.

Returns

VoidT. On error, `SRW_errno` is set to one of the following error codes:

- `SRW_E_SUCCESS`
- `SRW_E_BAD_VALUE`
- `SRW_E_WRITE_DTDS`
- `SRW_E_WRITE_INST`

Examples

The following code writes an attribute specification to the markup document instance:

```
. . .
StringT attributeName, attributeValue;
. . .
Sw_WriteAttrSpec(SW_LOC_INSTANCE, attributeName,
    attributeValue);
. . .
```

See also

- [“Sw_WriteDelimiter\(\)” on page 377](#)
- [“Sw_WriteReservedName\(\)” on page 379](#)
- [“Sw_WriteString\(\)” on page 380](#)
- [“SwLocationT” on page 406](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `StringT` and `VoidT`

Sw_WriteDelimiter()

Writes the specified delimiter string to the DTD or to the markup document instance.

Synopsis

```
#include "fm_struct.h"
. . .
VoidT Sw_WriteDelimiter(SwLocationT loc,
    StructuredDelimiterTypeT delType);
```

Arguments

<i>loc</i>	SW_LOC_DTD_SUBSET to write to the DTD, or SW_LOC_INSTANCE to write to the markup document instance
<i>delType</i>	Delimiter type to write

Details

To write a specific delimiter string to the DTD or to the markup document instance, call `Sw_WriteDelimiter()`. The *loc* argument specifies where to write.

Important: If you write directly to the DTD or document instance, you should not pass the event/conversion object pair to the default conversion routine. Instead, you should notify `FrameMaker` that you wrote out a value and return control to `FrameMaker`, so that it can pass the next event/conversion pair to your custom handler.

This function can be called only after an `SW_EVT_BEGIN_DOC` or `SW_EVT_BEGIN_BOOK_COMP` event has occurred and before the corresponding `SW_EVT_END_DOC` or `SW_EVT_END_BOOK_COMP` event.

Returns

`VoidT`. On error, `SRW_errno` is set to one of the following error codes:

- `SRW_E_SUCCESS`
- `SRW_E_BAD_VALUE`
- `SRW_E_WRITE_DTDS`
- `SRW_E_WRITE_INST`

Examples

The following code writes a delimiter to the DTD subset:

```
. . .
StructuredDelimiterTypeT delimType;
. . .
Sw_WriteDelimiter(SW_LOC_DTD_SUBSET, delimType);
. . .
```

See also

- [“Sw_WriteAttrSpec\(\)” on page 376](#)
- [“Sw_WriteReservedName\(\)” on page 379](#)
- [“Sw_WriteString\(\)” on page 380](#)
- [“SwLocationT” on page 406](#)
- [“StructuredDelimiterTypeT” on page 387](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `VoidT`

Sw_WriteReservedName()

Writes a reserved name to the DTD or to the markup document instance.

Synopsis

```
#include "fm_struct.h"

. . .

VoidT Sw_WriteReservedName(SwLocationT loc,
    StructuredReservedNameT rName);
```

Arguments

<i>loc</i>	SW_LOC_DTD_SUBSET to write to the DTD, or SW_LOC_INSTANCE to write to the markup document instance.
<i>rName</i>	Reserved name to write

Details

To write a reserved name to a DTD or to the markup document instance, call `Sw_WriteReservedName()`. The *loc* argument specifies the location to which to write.

Important: If you write directly to the DTD or document instance, you should not pass the event/conversion object pair to the default conversion routine. Instead, you notify FrameMaker that you wrote out a value, and return control to FrameMaker, so that it can pass the next event/conversion pair to your custom handler.

This function can be called only after an SW_EVT_BEGIN_DOC or SW_EVT_BEGIN_BOOK_COMP event has occurred and before the corresponding SW_EVT_END_DOC or SW_EVT_END_BOOK_COMP event.

Returns

VoidT. On error, SRW_errno is set to one of the following error codes:

- SRW_E_SUCCESS
- SRW_E_BAD_VALUE
- SRW_E_WRITE_DTDS
- SRW_E_WRITE_INST

Examples

The following code writes a reserved name to the markup document instance:

```
. . .
StructuredReservedNameT reservedNameType;
. . .
Sw_WriteReservedName(SW_LOC_INSTANCE, reservedNameType);
. . .
```

See also

- [“Sw_WriteAttrSpec\(\)” on page 376](#)
- [“Sw_WriteDelimiter\(\)” on page 377](#)
- [“Sw_WriteString\(\)” on page 380](#)
- [“SwLocationT” on page 406](#)
- [“StructuredReservedNameT” on page 392](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on `VoidT`

Sw_WriteString()

Writes the specified string to the specified location in the DTD or the markup document instance.

Synopsis

```
#include "fm_struct.h"
. . .
VoidT Sw_WriteString(SwLocationT loc,StringT s);
```

Arguments

<i>loc</i>	SW_LOC_DTD_SUBSET to write to the DTD, or SW_LOC_INSTANCE to write to the markup document instance
<i>s</i>	The string to write

Details

To write a string to a DTD or to a markup document instance, call `Sw_WriteString()`. The *loc* argument specifies where to write the string. This function does not convert any character references.

Important: If you write directly to the DTD or document instance, you should not pass the event/conversion object pair to the default conversion routine. Instead, you should notify FrameMaker that you wrote out a value, and return control to FrameMaker, so that it can pass the next event/conversion pair to your custom handler.

This function can be called only after an `SW_EVT_BEGIN_DOC` or `SW_EVT_BEGIN_BOOK_COMP` event has occurred and before the corresponding `SW_EVT_END_DOC` or `SW_EVT_END_BOOK_COMP` event.

Returns

VoidT. On error, SRW_errno is set to one of the following error codes:

- SRW_E_SUCCESS
- SRW_E_BAD_VALUE
- SRW_E_WRITE_DTDS
- SRW_E_WRITE_INST

Examples

The following code writes a string to the DTD:

```
. . .
StringT textToWrite;
. . .
Sw_WriteString(SW_LOC_DTD_SUBSET, textToWrite);
. . .
```

See also

- [“Sw_WriteAttrSpec\(\)” on page 376](#)
- [“Sw_WriteDelimiter\(\)” on page 377](#)
- [“Sw_WriteReservedName\(\)” on page 379](#)
- [“SwLocationT” on page 406](#)
- [“SRW_errno” on page 418](#)
- [“Primitive data types” on page 383](#) for more information on StringT and VoidT

Sw_WriteString()

5

Data Types and Structures Reference

This chapter describes the data types and structures used by structure import/export API functions. It contains these sections:

- [“Primitive data types,” next](#)
- [“Enumerated data types” on page 383](#)
- [“Data structures” on page 409](#)

Primitive data types

The following table lists the subset of primitive FDE data types used in FrameMaker and the structure import/export API. Primitive data types provide platform-independent versions of C data types. They are used to define all FDK structures and API functions. For a list of all the data types provided by the FDE, see Chapter 15, “Introduction to the FDE,” in the *FDK Programmer’s Guide*. When you include `fm_struct.h` in your source files, these primitive data types are automatically defined for you. You can use them to define function arguments and return types.

Frame API data type	Equivalent fundamental type	Size
BoolT	long	Signed 4 bytes
ConStringT	const unsigned char*	Pointer to const unsigned characters
F_ObjHandleT	long	Unsigned 4 bytes
IntT	long	Signed 4 bytes
StringT	unsigned char*	Pointer
UCharT	unsigned char	Unsigned 1 byte
PUCharT	unsigned int	Unsigned 4 bytes
UIntT	unsigned long	Unsigned 4 bytes
VoidT	void	None
PtrT	void*	Pointer

Enumerated data types

Enumerated data types are groups of consecutive integers that represent a specific range of possible values for function arguments or return types. Each enumeration in a data type is assigned an individual name and can be addressed by that name. Using enumerated data type names in code makes reading and maintaining the code easier.

The structure import/export API functions frequently expect enumerated data types as arguments, or return them as values. For example, many functions return error codes that indicate the success or failure of the operation performed by the function. The `SrWErrorT` enumerated data type declaration lists the error codes used with the structure import/export API.

This section describes the enumerated data types declared for and used with the structure import/export API. For each data type, it lists each value by name, describes what it indicates, and cross- references the structure import/export API data structures and functions that use the data type.

StructuredAttrDeclaredValueT

Deprecated: SgmlAttrDeclaredValueT

In an a DTD the declared value parameter of an attribute definition determines the type of value that must occur when the attribute is specified. In the structure import/export API, these types are enumerated in `StructuredAttrDeclaredValueT`.

Value	Meaning
STRUCTURED_AT_CDATA	Attribute value is character data (<i>CDATA</i>)
STRUCTURED_AT_ENTITY	Attribute value is general entity name (<i>ENTITY</i>)
STRUCTURED_AT_ENTITIES	Attribute value is general entity name list (<i>ENTITIES</i>)
STRUCTURED_AT_ID	Attribute value is ID value (<i>ID</i>)
STRUCTURED_AT_IDREF	Attribute value is ID reference value (<i>IDREF</i>)
STRUCTURED_AT_IDREFS	Attribute value is ID reference list (<i>IDREFS</i>)
STRUCTURED_AT_NAME	Attribute value is name (<i>NAME</i>)
STRUCTURED_AT_NAMES	Attribute value is name list (<i>NAMES</i>)
STRUCTURED_AT_NMTOKEN	Attribute value is name token (<i>NMTOKEN</i>)
STRUCTURED_AT_NMTOKENS	Attribute value is name token list (<i>NMTOKENS</i>)
STRUCTURED_AT_NOTATION	Attribute value is notation name (<i>NOTATION</i>)
STRUCTURED_AT_NUMBER	Attribute value is number (<i>NUMBER</i>)
STRUCTURED_AT_NUMBERS	Attribute value is number list (<i>NUMBERS</i>)
STRUCTURED_AT_NUTOKEN	Attribute value is number token (<i>NUTOKEN</i>)
STRUCTURED_AT_NUTOKENS	Attribute value is number token list (<i>NUTOKENS</i>)
STRUCTURED_AT_NMTOKENGRP	Attribute value is name-token group

Enumerations correspond directly to reserved words (in parentheses in the meaning column), except for `STRUCTURED_AT_NMTOKENGRP`, which indicates that the declared value is a name token group. `StructuredAttrDeclaredValueT` is a data field in the attribute definition structure `StructuredAttrDefT`.

See also

- [“StructuredAttrDefT” on page 411](#)
- [“StructuredAttrDefaultValueT” on page 385](#)
- [“StructuredReservedNameT” on page 392](#)

StructuredAttrDefaultValueT

Deprecated: SgmlAttrDefaultValueT

In the DTD, the default value parameter of an attribute definition determines whether an attribute value must be specified or can be omitted. It also sets the value that is used for an attribute if a value is not specified. In the structure import/export API, these types are enumerated in `StructuredAttrDefaultValueT`.

Value	Meaning
STRUCTURED_AV_FIXED	Attribute is fixed (<i>FIXED</i>)
STRUCTURED_AV_REQUIRED	Attribute is required (<i>REQUIRED</i>)
STRUCTURED_AV_CURRENT	Attribute is current (<i>CURRENT</i>)
STRUCTURED_AV_CONREF	Attribute is content reference (<i>CONREF</i>)
STRUCTURED_AV IMPLIED	Attribute is implied (<i>IMPLIED</i>)
STRUCTURED_AV_DEFAULT	Attribute has a default value (<i>DEFAULT</i>)

Enumerations correspond directly to reserved words (in parentheses in the meaning column).

`StructuredAttrDefaultValueT` is a data field in the attribute definition structure `StructuredAttrDefT`.

See also

- [“StructuredAttrDefT” on page 411](#)
- [“StructuredAttrDeclaredValueT” on page 384](#)
- [“StructuredReservedNameT” on page 392](#)

StructuredContentTokenT

Deprecated: SgmlContentTokenT

In the DTD, an element definition includes a model group, tokens that represent elements, and connectors that indicate the relationship among tokens. In the structure import/export API, possible tokens in a model group are enumerated in `StructuredContentTokenT`.

Value	Meaning
STRUCTURED_CT_GRP_O	Group open
STRUCTURED_CT_GRP_C	Group close

Value	Meaning
STRUCTURED_CT_PCDATA	Parsed character data (<i>#PCDATA</i>)
STRUCTURED_CT_REP	Content optional and repeatable zero or more times
STRUCTURED_CT_PLUS	Content required and repeatable one or more times
STRUCTURED_CT_OPT	Content can appear only once or can be omitted entirely
STRUCTURED_CT_SEQ	All content required in order specified by tokens
STRUCTURED_CT_AND	Content must appear, but can appear in any order
STRUCTURED_CT_OR	Only one element or data instance must occur
STRUCTURED_CT_NAME	Element name

In the structure import/export API, a model group is an array containing enumerated types from `StructuredContentTokenT`. If a model group contains an occurrence of `STRUCTURED_CT_NAME`, the next item in the array contains the length of the represented name, *n*. The next *n* items in the array contain one character in sequence from the element name. There is no terminating `NULL` byte for the element name.

`StructuredContentTokenT` is a data field in the element definition structure `StructuredElementDefT`.

See also

- [“StructuredElementDefT” on page 413](#)
- [“StructuredAttrDefsT” on page 411](#)
- [“StructuredContentTypeT” on page 386](#)

StructuredContentTypeT

Deprecated: SgmlContentTypeT

In the DTD, an element declaration can declare actual content or a content model. In the FrameMaker API, the content type of an element declaration is enumerated in `StructuredContentTypeT`.

Value	Meaning
STRUCTURED_EC_MODEL_GROUP	Declared content is a model group
STRUCTURED_EC_ANY	Declared content is <i>ANY</i>
STRUCTURED_EC_EMPTY	Declared content is <i>EMPTY</i>
STRUCTURED_EC_CDATA	Declared content is <i>CDATA</i>
STRUCTURED_EC_RCDATA	Declared content is <i>RCDATA</i>

If an element declaration is a content model, the declaration indicates whether the content model has a model group (`STRUCTURED_EC_MODEL_GROUP`) or the *ANY* reserved name (`STRUCTURED_EC_ANY`). Otherwise, actual content is declared (`STRUCTURED_EC_EMPTY`, `STRUCTURED_EC_CDATA`, or `STRUCTURED_EC_RCDATA`).

StructuredContentTypeT is a data field in the element definition structure StructuredElementDefT.

See also

- [“StructuredElementDefT” on page 413](#)
- [“StructuredAttrDefsT” on page 411](#)
- [“StructuredContentTokenT” on page 385](#)

StructuredDelimiterTypeT

Deprecated: SgmlDelimiterTypeT

In the SGML declaration, delimiters are used to distinguish between markup and data. In the FrameMaker API, delimiters are enumerated in StructuredDelimiterTypeT.

Value	Meaning
STRUCTURED_DE_AND	AND connector
STRUCTURED_DE_COM	Comment start or end
STRUCTURED_DE_CRO	Character reference open
STRUCTURED_DE_DSC	Declaration subset close
STRUCTURED_DE_DSO	Declaration subset open
STRUCTURED_DE_DTGC	Data-tag group close
STRUCTURED_DE_DTGO	Data-tag group open
STRUCTURED_DE_ERO	Entity reference open
STRUCTURED_DE_ETAGO	End-tag open
STRUCTURED_DE_GRPCL	Group close
STRUCTURED_DE_GRPOL	Group open
STRUCTURED_DE_LIT	Literal start or end
STRUCTURED_DE_LITA	Literal start or end (alternative)
STRUCTURED_DE_MDC	Markup declaration close
STRUCTURED_DE_MDO	Markup declaration open
STRUCTURED_DE_MINUS	Exclusion
STRUCTURED_DE_MSC	Marked-section close
STRUCTURED_DE_NET	Null end-tag
STRUCTURED_DE_OPT	Optional occurrence indicator
STRUCTURED_DE_OR	OR connector
STRUCTURED_DE_PERO	Parameter entity reference open
STRUCTURED_DE_PIC	Processing instruction close
STRUCTURED_DE_PIO	Processing instruction open

Value	Meaning
STRUCTURED_DE_PLUS	Required and repeatable; inclusion
STRUCTURED_DE_REFC	Reference close
STRUCTURED_DE_REP	Optional and repeatable
STRUCTURED_DE_RNI	Reserved name indicator
STRUCTURED_DE_SEQ	Sequence connector
STRUCTURED_DE_STAGO	Start-tag open
STRUCTURED_DE_TAGC	Tag close
STRUCTURED_DE_VI	Value indicator

StructuredDelimiterTypeT is passed as an argument to Structured_GetDelimiterString().

See also

- [“Structured_GetDelimiterString\(\)” on page 59](#)
- [“StructuredQuantityTypeT” on page 391](#)
- [“StructuredReservedNameT” on page 392](#)
- [“StructuredFeatureTypeT” on page 390](#)

StructuredEntityScopeT

Deprecated: SgmlEntityScopeT

In the DTD, the role played by an entity is called its scope. In FrameMaker, entity scope is enumerated in StructuredEntityScopeT.

Value	Meaning
STRUCTURED_ES_GENERAL	General entity
STRUCTURED_ES_PARAMETER	Parameter entity

If StructuredEntityScopeT is STRUCTURED_ES_PARAMETER, the entity name is passed as an argument or returned as a value without the initial parameter open (*PERO*) delimiter.

StructuredEntityScopeT is a data field in the entity definition structure StructuredEntityDefT. It is also passed as an argument to both Structured_GetFirstEntityName() and Structured_GetNextEntityName().

See also

- [“Structured_GetFirstEntityName\(\)” on page 68](#)
- [“Structured_GetNextEntityName\(\)” on page 78](#)
- [“StructuredEntityDefT” on page 414](#)

- [“StructuredEntityTypeT” on page 389](#)

StructuredEntityTypeT

Deprecated: SgmlEntityTypeT

In the DTD, entities are differentiated by the type of information they contain. In FrameMaker, entity types are enumerated in `StructuredEntityTypeT`.

Value	Meaning
STRUCTURED_ET_TEXT	Entity is nonbracketed text
STRUCTURED_ET_CDATA	Entity is <i>CDATA</i>
STRUCTURED_ET_SDATA	Entity is <i>SDATA</i>
STRUCTURED_ET_NDATA	Entity is <i>NDATA</i>
STRUCTURED_ET_PI	Entity is a processing instruction
STRUCTURED_ET_STARTTAG	Entity is a start-tag
STRUCTURED_ET_ENDTAG	Entity is an end-tag
STRUCTURED_ET_MS	Entity is bracketed text for a marked section
STRUCTURED_ET_MD	Entity is bracketed text for a markup declaration
STRUCTURED_ET_SUBDOC	Entity is a subdocument

`STRUCTURED_ET_MD` is used for bracketed text that is a markup declaration. It is preceded by a markup declaration open (*MDO*) delimiter, and followed by a markup declaration close (*MDC*) delimiter.

`STRUCTURED_ET_TEXT` identifies an entity that is not a data entity or subdocument entity.

Other types distinguish various types of bracketed text, data entities, and subdocument entities.

`StructuredEntityTypeT` is a data field in the entity definition structure `StructuredEntityDefT`. `StructuredEntityDefT` structures are returned by `Structured_GetEntityDef()` and `Structured_GetDefaultEntityDef()`.

See also

- [“Structured_GetDefaultEntityDef\(\)” on page 58](#)
- [“Structured_GetEntityDef\(\)” on page 63](#)
- [“StructuredEntityDefT” on page 414](#)
- [“StructuredEntityScopeT” on page 388](#)

StructuredFeatureTypeT

Deprecated: SgmlFeatureTypeT

An SGML declaration can identify optional SGML features that it supports. In FrameMaker, optional features are enumerated in `StructuredFeatureTypeT`.

Value	Meaning
STRUCTURED_FEAT_SHORTREF	Short entity reference delimiters (<i>SHORTREF</i>)
STRUCTURED_FEAT_DATATAG	Data-tag minimization (<i>DATATAG</i>)
STRUCTURED_FEAT_OMITTAG	Omitted tag minimization (<i>OMITTAG</i>)
STRUCTURED_FEAT_RANK	Omitted rank suffix minimization (<i>RANK</i>)
STRUCTURED_FEAT_SHORTTAG	Short tag minimization (<i>SHORTTAG</i>)
STRUCTURED_FEAT_SIMPLE_LINK	Simple link process (<i>SIMPLE</i>)
STRUCTURED_FEAT_IMPLICIT_LINK	Implicit link process (<i>IMPLICIT</i>)
STRUCTURED_FEAT_EXPLICIT_LINK	Explicit link process (<i>EXPLICIT</i>)
STRUCTURED_FEAT_CONCUR	Concurrent document type instances (<i>CONCUR</i>)
STRUCTURED_FEAT_SUBDOC	Nested subdocuments (<i>SUBDOC</i>)
STRUCTURED_FEAT_FORMAL	Formal public identifiers (<i>FORMAL</i>)

While FrameMaker can test for optional features in a declaration, it only processes the following subset of optional features:

- *SHORTTAG*
- *SHORTREF*
- *OMITTAG*
- *FORMAL*

If during processing FrameMaker encounters an optional feature it does not support, an error is issued, and processing is terminated.

`StructuredFeatureTypeT` is returned by `Structured_GetOptionalFeature()`, which tests an SGML declaration for the presence of optional features.

See also

- [“Structured_GetOptionalFeature\(\)” on page 83](#)
- [“StructuredDelimiterTypeT” on page 387](#)
- [“StructuredQuantityTypeT” on page 391](#)
- [“StructuredReservedNameT” on page 392](#)

StructuredQuantityTypeT

Deprecated: SgmlQuantityTypeTSgmlQuantityTypeT

An SGML declaration can contain information about the maximum number, or maximum length, of attributes, entities, and elements. These maximum values are called quantity limits. In FrameMaker, quantity limits are enumerated in `StructuredQuantityTypeT`.

Value	Meaning
STRUCTURED_QTY_ATT CNT	Maximum number of attribute names and name tokens in an attribute definition list
STRUCTURED_QTY_ATT SPLEN	Maximum normalized length of an attribute specification list for a start-tag
STRUCTURED_QTY_BSEQLEN	Maximum length of a blank sequence in a short reference string
STRUCTURED_QTY_DTAGLEN	Maximum length of a data tag
STRUCTURED_QTY_DTEMPL EN	Maximum length of a data-tag template or pattern template
STRUCTURED_QTY_ENTLVL	Maximum nesting level for entities
STRUCTURED_QTY_GRP CNT	Maximum number of tokens in a group
STRUCTURED_QTY_GRP GTCNT	Maximum number of content tokens at all levels of a content model
STRUCTURED_QTY_GRP LVL	Maximum nesting level of model groups
STRUCTURED_QTY_LITLEN	Maximum length of a parameter literal or an attribute literal
STRUCTURED_QTY_NAMELEN	Maximum length of a name, a name token, a number, or other item
STRUCTURED_QTY_NORMSEP	Length to use instead of counting separators when calculating normalized lengths
STRUCTURED_QTY_PILEN	Maximum length of a processing instruction
STRUCTURED_QTY_TAGLEN	Maximum length of a start-tag
STRUCTURED_QTY_TAGLVL	Maximum nesting level of open elements

You pass `StructuredQuantityTypeT` as an argument to `Structured_GetQuantity()`.

See also

- [“Structured_GetQuantity\(\)” on page 85](#)
- [“StructuredDelimiterTypeT” on page 387](#)
- [“StructuredFeatureTypeT” on page 390](#)
- [“StructuredReservedNameT” on page 392](#)

StructuredReservedNameT

Deprecated: SgmlReservedNameT

Reserved names are defined by the reference concrete syntax, or in the SGML declaration. In FrameMaker, reserved names are enumerated in `StructuredReservedNameT`. The following values are enumerated for `StructuredReservedNameT`—entries with a "†" indicate values that are valid for both SGML and XML:

STRUCTURED_RN_ANY †	STRUCTURED_RN_INCLUDE †	STRUCTURED_RN_POSTLINK
STRUCTURED_RN_ATTLIST †	STRUCTURED_RN_INITIAL	STRUCTURED_RN_PUBLIC †
STRUCTURED_RN_CDATA †	STRUCTURED_RN_LINK	STRUCTURED_RN_RCDATA
STRUCTURED_RN_CONREF	STRUCTURED_RN_LINKTYPE	STRUCTURED_RN_RE
STRUCTURED_RN_CURRENT	STRUCTURED_RN_MD	STRUCTURED_RN_REQUIRED †
STRUCTURED_RN_DEFAULT	STRUCTURED_RN_MS	STRUCTURED_RN_RESTORE
STRUCTURED_RN_DOCTYPE †	STRUCTURED_RN_NAME	STRUCTURED_RN_RS
STRUCTURED_RN_ELEMENT †	STRUCTURED_RN_NAMES	STRUCTURED_RN_SDATA
STRUCTURED_RN_EMPTY †	STRUCTURED_RN_NDATA †	STRUCTURED_RN_SHORTREF
STRUCTURED_RN_ENDTAG	STRUCTURED_RN_NMTOKEN †	STRUCTURED_RN_SIMPLE
STRUCTURED_RN_ENTITIES †	STRUCTURED_RN_NMTOKENS †	STRUCTURED_RN_SPACE
STRUCTURED_RN_ENTITY †	STRUCTURED_RN_NOTATION †	STRUCTURED_RN_STARTTAG
STRUCTURED_RN_FIXED †	STRUCTURED_RN_NUMBER	STRUCTURED_RN_SUBDOC
STRUCTURED_RN_ID †	STRUCTURED_RN_NUMBERS	STRUCTURED_RN_SYSTEM †
STRUCTURED_RN_IDLINK	STRUCTURED_RN_NUTOKEN	STRUCTURED_RN_TEMP
STRUCTURED_RN_IDREF †	STRUCTURED_RN_NUTOKENS	STRUCTURED_RN_USELINK
STRUCTURED_RN_IDREFS †	STRUCTURED_RN_O	STRUCTURED_RN_USEMAP
STRUCTURED_RN_IGNORE †	STRUCTURED_RN_PCDATA †	
STRUCTURED_RN_IMPLIED †	STRUCTURED_RN_PI	

Each entry in `StructuredReservedNameT` corresponds directly to a reserved name. To discover the reserved name that applies to an enumerated type in `StructuredReservedNameT`, examine the last word in the enumerated type name. For example, `STRUCTURED_RN_ANY` refers to the reserved name `ANY`, and the reserved name, `STRUCTURED_RN_ATTLIST` refers to `ATTLIST`.

`StructuredReservedNameT` is returned by `Structured_GetReservedName()`.

See also

- [“Structured_GetReservedName\(\)” on page 87](#)
- [“StructuredDelimiterTypeT” on page 387](#)
- [“StructuredFeatureTypeT” on page 390](#)

- [“StructuredQuantityTypeT” on page 391](#)

SrEventTypeT

FrameMaker imports a markup document by creating events and conversion objects for each declaration or statement it encounters in the markup document. It also generates events to signal the start or end of importing, or to signal changes in the status of the FrameMaker document receiving the imported markup document. The types of events that can be created on import are enumerated in `SrEventTypeT`.

Event	Meaning
SR_EVT_BEGIN_READER	Start of import processing. An import/export client can respond to this event by doing any necessary setup work, such as processing based on settings in the SGML declaration, before the document itself is imported.
SR_EVT_END_READER	End of import processing. An import/export client can respond to this event by doing any desired cleanup work after all of a document's events are processed.
SR_EVT_BEGIN_BOOK	Start of FrameMaker book creation.
SR_EVT_END_BOOK	End of import into FrameMaker book. When this event occurs, FrameMaker applies format rules to and removes overrides from every book component. To create a document with format overrides, save override information during processing, and apply it as the last step of handling this event.
SR_EVT_BEGIN_BOOK_COMP	Start of FrameMaker document that is part of a book. This event occurs instead of <code>SR_EVT_BEGIN_DOC</code> for documents that are part of a book.
SR_EVT_END_BOOK_COMP	End of FrameMaker document that is part of a book. This event occurs instead of <code>SR_EVT_END_DOC</code> for documents that are part of a book.
SR_EVT_BEGIN_DOC	Start of an individual FrameMaker document that is <i>not</i> part of a book.
SR_EVT_END_DOC	End of an individual FrameMaker document that is <i>not</i> part of a book. When this event occurs, FrameMaker applies format rules and removes overrides. To create a document with format overrides, save override information during processing, and apply it as the last step of handling this event.
SR_EVT_BEGIN_ELEM	Start of an element in markup.

Event	Meaning
SR_EVT_END_ELEM	End of an element in markup. Elements with declared content of <i>EMPTY</i> do not trigger this event.
SR_EVT_BEGIN_ENTITY	Start of an entity in markup.
SR_EVT_END_ENTITY	End of an entity in markup.
SR_EVT_RE	Data record end that is not ignored.
SR_EVT_PI	Processing instruction encountered.
SR_EVT_CDATA	Sequence of data characters (excluding record end) occurring in <i>CDATA</i> or <i>PCDATA</i> . Replacement text for an entity reference may signal one or more separate events. Replacement text for a character reference signals only one event.

SrEventTypeT is a data field in the import event structure SrEventT. SrEventT is passed as an argument to Sr_Convert() and Sr_EventHandler(). SrEventT is returned by Sr_GetAssociatedEvent().

See also

- [“Sr_Convert\(\)” on page 104](#)
- [“Sr_EventHandler\(\)” on page 107](#)
- [“Sr_GetAssociatedEvent\(\)” on page 109](#)
- [“SrEventT” on page 416](#)
- [“SrObjTypeT” on page 395](#)
- [“SrObjTypeT” on page 395](#)

SrLocationT

When FrameMaker proposes to insert an object in a FrameMaker document, it also proposes a general insertion location for the object. General insertion locations are enumerated in SrLocationT.

Value	Meaning
SR_LOC_FLOW	Insert into a FrameMaker text flow
SR_LOC_BOOK	Insert into a FrameMaker book
SR_LOC_ELEMENT	Insert into a document at a specific location
SR_LOC_MARKER_TEXT	Insert into a specific marker
SR_LOC_TEXT_INSET	Insert into a text inset

SrLocationT is a data field in the FrameMaker insertion location structure, SrInsertLocT. SrInsertLocT is returned by Sr_GetInsertLoc() and is passed as an argument to Sr_SetInsertLoc().

See also

- [“Sr_GetInsertLoc\(\)” on page 147](#)
- [“Sr_SetInsertLoc\(\)” on page 210](#)
- [“SrInsertLocT” on page 416](#)

SrObjTypeT

FrameMaker imports a markup document by creating events and conversion objects for each declaration or statement it encounters in the markup document. The types of import conversion objects that FrameMaker generates are enumerated in SrObjTypeT.

Value	Meaning
SR_OBJ_SESSION	Private, client-provided session data
SR_OBJ_BOOK	FrameMaker book
SR_OBJ_BOOK_COMP	FrameMaker book component
SR_OBJ_DOC	FrameMaker document
SR_OBJ_ELEM	FrameMaker element
SR_OBJ_TABLE	FrameMaker table
SR_OBJ_TABLE_TITLE	FrameMaker table title
SR_OBJ_TABLE_HEADING	FrameMaker table heading
SR_OBJ_TABLE_BODY	FrameMaker table body
SR_OBJ_TABLE_FOOTING	FrameMaker table footing
SR_OBJ_TABLE_ROW	Row in a FrameMaker table
SR_OBJ_TABLE_CELL	Cell in a FrameMaker table
SR_OBJ_COLSPEC	CALS table <i>colspec</i>
SR_OBJ_SPANSPEC	CALS table <i>spanspec</i>
SR_OBJ_GRAPHIC	Imported graphic or FrameMaker anchored frame
SR_OBJ_EQUATION	FrameMaker equation
SR_OBJ_VARIABLE	FrameMaker variable
SR_OBJ_MARKER	FrameMaker marker
SR_OBJ_XREF	FrameMaker cross-reference
SR_OBJ_FOOTNOTE	FrameMaker footnote
SR_OBJ_TEXT	Text segment
SR_OBJ_SPECIAL_CHAR	Special FrameMaker character mapped to an <i>SDATA</i> entity
SR_OBJ_TEXT_INSET	FrameMaker text inset

Value	Meaning
SR_OBJ_REF_ELEM	FrameMaker reference element mapped to an <i>SDATA</i> entity (stored on the reference page of the current FrameMaker template document)
SR_OBJ_RUBI_GROUP	FrameMaker Rubi group
SR_OBJ_RUBI	FrameMaker Rubi object
SR_OBJ_ENTITY	Any FrameMaker object that is created as the result of an entity in markup

SrObjTypeT is a data field in an import conversion object structure pointed to by SrConvObjT. An import conversion object's internal structure can differ depending on its object type. Import/export client applications should never directly access an import conversion object. Instead, the client should use structure import/export API calls to return conversion object information.

SrObjTypeT is returned by the `Sr_GetObjType()` function, which is called to determine the object's type. For an example, see `Sr_EventHandler()`.

See also

- [“Sr_EventHandler\(\)” on page 107](#)
- [“Sr_GetObjType\(\)” on page 152](#)
- [“SrConvObjT” on page 415](#)

SrwErrorT

When most FrameMaker functions execute, they return or set an integer error status code. FrameMaker error codes are enumerated in `SrwErrorT`.

Value	Meaning
SRW_E_SUCCESS	No error occurred
SRW_E_FAILURE	Function cannot perform requested operation
SRW_E_INVALID_CONV_OBJ	Improperly constructed conversion object passed as a function argument
SRW_E_WRONG_OBJ_TYPE	Conversion object specified for an inappropriate object type
SRW_E_OBJ_HAS_NO_SUCH_PROP	Conversion object does not have the specified formatting property
SRW_E_NO_SUCH_ATTR	Requested attribute does not exist
SRW_E_NOT_CUR_DOC_ID	Object ID is not the object ID of the current document
SRW_E_BAD_OBJ_HANDLE	Incorrect object ID passed

Value	Meaning
SRW_E_NO_TEMPLATE	Information was requested from the current import template, but the current structure application does not specify a FrameMaker template for import
SRW_E_BAD_VALUE	Illegal value passed as a function argument
SRW_E_NOT_BOOK_COMP	Book component function attempted to operate on a document that is not a book component
SRW_E_INVALID_TEXT_ITEM_TYPE	Attempt to create a markup text item with an invalid item type
SRW_E_BAD_ROW_SCAN_ORDER	Cannot scan rows in the order specified
SRW_E_WRITE_INST	FrameMaker could not write information to the document instance
SRW_E_WRITE_DTDS	FrameMaker could not write information to the DTD subset

Many structure import/export API functions specifically return a type of `SrwErrorT`. Many other API functions return other types, but also set the global error variable `SRW_errno` to an `SrwErrorT` type.

See also

- [“SrwErrorT” on page 396](#)
- [“Primitive data types” on page 383](#) for information on `IntT`

SrwFmPropertyT

`SrwFmPropertyT` is an enumerated data field in `SrwPropValT` that describes the type of information contained in that structure. Because there are so many property types, they are subdivided according to type of FrameMaker object in the following tables.

The following table lists the table properties in `SrwFmPropertyT`:

Value	Meaning
SRW_PROP_COLUMNS	Number of columns in table
SRW_PROP_COLUMN_WIDTHS	Widths of columns in table
SRW_PROP_TABLE_FORMAT	Table format
SRW_PROP_COLUMN_FORMATS	Column format
SRW_PROP_PGF_FORMAT	Paragraph format for a specific cell
SRW_PROP_MINIMUM_HEIGHT	Minimum height for a specific row
SRW_PROP_MAXIMUM_HEIGHT	Maximum height for a specific row
SRW_PROP_HSTRAD	Cell connections for a horizontal straddle
SRW_PROP_VSTRAD	Cell connections for a vertical straddle

Value	Meaning
SRW_PROP_MORE_ROWS	Number of rows in a vertical straddle
SRW_PROP_TABLE_BORDER	Table border ruling*
SRW_PROP_ROW_TYPE	Type of row*
SRW_PROP_VSTRAD_START	First cell in a vertical straddle
SRW_PROP_VSTRAD_END_AT	Last cell in a vertical straddle
STW_PROP_VSTRAD_END_BEFORE	First cell following a vertical straddle
SRW_PROP_PAGE_WIDE	Width of table on a page
SRW_PROP_ROTATE	Rotate table
SRW_PROP_INSERT_TITLE	Insert a table title
SRW_PROP_INSERT_HEADING	Insert heading row(s)
SRW_PROP_INSERT_FOOTING	Insert footing rows(s)

*On import into FrameMaker, values for `SRW_PROP_TABLE_BORDER` and `SRW_PROP_ROW_TYPE` are enumerated types as specified in `SrwFmPropValT`.

The following table lists the *CALS* table properties in `SrwFmPropertyT`:

Value	Meaning
SRW_PROP_COL_NAME	<i>Colspec</i> name
SRW_PROP_COL_NUM	<i>Colspec</i> number
SRW_PROP_COL_WIDTH	<i>Colspec</i> width
SRW_PROP_SPAN_NAME	<i>Spanspec</i> name
SRW_PROP_NAME_START	<i>Spanspec</i> starting location
SRW_PROP_NAME_END	<i>Spanspec</i> ending location
SRW_PROP_COL_RULING	<i>Colspec</i> ruling
SRW_PROP_ROW_RULING	Row ruling
SRW_PROP_ALIGN_TYPE	Alignment type
SRW_PROP_ALIGN_CHAR	Alignment character
SRW_PROP_ALIGN_OFFSET	Alignment offset

The following table lists the graphics properties in `SrwFmPropertyT`:

Value	Meaning
SRW_PROP_ENTITY	Entity definition to store with graphic
SRW_PROP_FILE	Filepath to store with graphic
SRW_PROP_DPI	Dots per inch to which to scale a bitmap graphic
SRW_PROP_IMPORT_SIZE	Width and height of the graphic in pixels
SRW_PROP_REF_OR_COPY	Import by reference or by copy*

Value	Meaning
SRW_PROP_SIDeways	Orientation of graphic about its vertical axis
SRW_PROP_IMPORT_ANGLE	Angle of graphic within its anchored frame
SRW_PROP_HOR_OFFSET	Horizontal offset of anchored frame relative to text column
SRW_PROP_VER_OFFSET	Vertical offset of anchored frame relative to text baseline
SRW_PROP_POSITION	Position of anchored frame*
SRW_PROP_ALIGNMENT	Alignment of anchored frame in text column*
SRW_PROP_CROPPED	Crop condition for anchored frame
SRW_PROP_FLOATING	Float condition for anchored frame
SRW_PROP_WIDTH	Width of anchored frame
SRW_PROP_HEIGHT	Height of anchored frame
SRW_PROP_ANGLE	Rotation angle of anchored frame
SRW_PROP_BLOFFSET	Anchored frame offset from paragraph baseline
SRW_PROP_NSOFFSET	Anchored frame offset from text column

*On import into FrameMaker, values for `SRW_PROP_REF_OR_COPY`, `SRW_PROP_POSITION`, and `SRW_PROP_ALIGNMENT` are enumerated types as specified in `SrwFmPropValT`.

The following table lists the XML export properties for exporting cross-references, graphics, and equations in FrameMaker versions earlier than 7.0. In these earlier versions of FrameMaker, the XML export process proposed values for these properties when appropriate. These properties are retained for backward compatibility:

Value	Meaning
SRW_PROP_XML_LINK	Value for the <code>xml:link</code> attribute
SRW_PROP_XML_HREF	Value for an <code>HREF</code> attribute
SRW_PROP_XML_SHOW	Value of a <code>SHOW</code> attribute: values can be <code>embed</code> , <code>new</code> , or <code>replace</code>
SRW_PROP_XML_ACTUATE	Value of an <code>ACTUATE</code> attribute: values can be <code>auto</code> or <code>user</code>

The following table lists the cross-reference properties in `SrwFmPropertyT`:

Value	Meaning
SRW_PROP_XREF_ID	Cross-reference ID
SRW_PROP_XREF_FORMAT	Cross-reference format

The following table lists the marker properties in `SrwFmPropertyT`:

Value	Meaning
SRW_PROP_MARKER_TYPE	Marker type

Value	Meaning
SRW_PROP_MARKER_TEXT	Marker content

FrameMaker conversion objects that can be mapped either to markup attributes or to FrameMaker formatting properties include tables, *CALS* tables, graphics, cross- references, and markers.

For example, consider a three-column table. In a markup document, the number of columns is expressed as an attribute. In a FrameMaker document, the number of columns is a property of the table. During the import/export conversion process, each piece of attribute/property information for a conversion object is stored in the property structure *SrwPropValT*. A second structure, *SrwPropValsT*, is attached to the conversion object. *SrwPropValsT* is a list of pointers to each *SrwPropValT* structure containing conversion object information.

See also

- [“SrwPropValT” on page 420](#)
- [“SrwPropValsT” on page 421](#)
- [“SrwFmPropValT” on page 401](#)

SrwFmPropValT

When FrameMaker proposes to map an import conversion object containing a table or graphic into a FrameMaker document, it may also propose table row, table border ruling, anchored frame alignment, or anchored frame position properties. These properties are enumerated in `SrwFmPropValT`.

Value	Meaning
<code>SRW_PVAL_ALEFT</code>	Align anchored frame left
<code>SRW_PVAL_ARIGHT</code>	Align anchored frame right
<code>SRW_PVAL_ACENTER</code>	Align anchored frame center
<code>SRW_PVAL_AINSIDE</code>	Align anchored frame on side closest to binding
<code>SRW_PVAL_AOUTSIDE</code>	Align anchored frame on side farthest from binding
<code>SRW_PVAL_INLINE</code>	Position anchored frame at insertion point
<code>SRW_PVAL_TOP</code>	Position anchored frame at top of column
<code>SRW_PVAL_BELOW</code>	Position anchored frame below current line
<code>SRW_PVAL_BOTTOM</code>	Position anchored frame at bottom of column
<code>SRW_PVAL_SC_LEFT</code>	Position anchored frame outside and left of column
<code>SRW_PVAL_SC_RIGHT</code>	Position anchored frame outside and right of column
<code>SRW_PVAL_SC_NEAREST</code>	Position anchored frame outside column closest to page edge
<code>SRW_PVAL_SC_FARTHEST</code>	Position anchored frame outside column farthest from page edge
<code>SRW_PVAL_SC_INSIDE</code>	Position anchored frame outside column closest to binding
<code>SRW_PVAL_SC_OUTSIDE</code>	Position anchored frame outside column farthest from binding
<code>SRW_PVAL_TF_LEFT</code>	Position anchored frame at left side of table cell
<code>SRW_PVAL_TF_RIGHT</code>	Position anchored frame at right side of table cell
<code>SRW_PVAL_TF_NEAREST</code>	Position anchored frame at side of table cell closest to page edge
<code>SRW_PVAL_TF_FARTHEST</code>	Position anchored frame at side of table cell farthest from page edge
<code>SRW_PVAL_TF_INSIDE</code>	Position anchored frame at side of table cell closest to binding
<code>SRW_PVAL_TF_OUTSIDE</code>	Position anchored frame at side of table cell farthest from binding
<code>SRW_PVAL_RUN_INTO_PGF</code>	Position anchored frame at left side of page frame
<code>SRW_PVAL_REFERENCE</code>	Import graphic by reference

Value	Meaning
SRW_PVAL_COPY	Import graphic by copy
SRW_PVAL_TOPBOT	Rule table top and bottom table borders
SRW_PVAL_SIDES	Rule table side borders
SRW_PVAL_ALL	Rule all table borders
SRW_PVAL_NONE	Do not rule any table borders
SRW_PVAL_HEADING	Table row is a heading
SRW_PVAL_BODY	Table row is a body row
SRW_PVAL_FOOTING	Table row is a footing

See also

- [“SrwFmPropertyT” on page 397](#)
- [“SrwPropValT” on page 420](#)
- [“SrwPropValsT” on page 421](#)

SrwLogDocT

When FrameMaker writes a message to a log file, it uses a message structure to contain the message information. Messages can refer to a FrameMaker document or to a text file (a markup instance). Message source documents are enumerated in `SrwLogDocT`.

Value	Meaning
SRW_LOGD_FILEPATH	The document is referenced by filepath
SRW_LOGD_ID	The document is referenced by ID

`SrwLogDocT` is a data field in the `SrwLogMessageLocationT` structure. `SrwLogMessageLocationT` is passed as an argument to `Srw_LogMessage()`.

See also

- [“Srw_LogMessage\(\)” on page 273](#)
- [“SrwLogLocT” on page 403](#)
- [“SrwLogMessageLocationT” on page 420](#)

SrwLogLocT

When FrameMaker writes a message to a log file, it uses a message structure to contain the message information. Messages can refer to different parts of a document, or to no document at all. Possible document reference locations are enumerated in `SrwLogLocT`.

Value	Meaning
<code>SRW_LOGL_NONE</code>	Log file message does not refer to a location within a file
<code>SRW_LOGL_ID</code>	Log file message is associated with the ID of a FrameMaker anchored frame, marker, cross-reference, paragraph, or element
<code>SRW_LOGL_LINE</code>	Log file message references a specific line in a document

For `SRW_LOGL_ID`, the cited FrameMaker document must be open, because the message will contain a hypertext link to the indicated object.

`SrwLogLocT` is a data field in `SrwLogMessageLocationT`. `SrwLogMessageLocationT` is passed as an argument to `Srw_LogMessage()`.

See also

- [“Srw_LogMessage\(\)” on page 273](#)
- [“SrwLogDocT” on page 402](#)
- [“SrwLogMessageLocationT” on page 420](#)

SrwTablePartTypeT

When FrameMaker creates table conversion objects, it specifies the type of rows in the conversion object. Possible row types are enumerated in `SrwTablePartTypeT`.

Value	Meaning
<code>SRW_TABLE_TITLE</code>	Table title
<code>SRW_TABLE_HEADING</code>	Table heading
<code>SRW_TABLE_BODY</code>	Table body
<code>SRW_TABLE_FOOTING</code>	Table footing

`SrwTablePartTypeT` is passed as an argument to `Sw_SetTableScanOrder()` and is returned by `Sr_GetRowType()`.

See also

- [“Sr_GetInsertedTablePartElementName\(\)” on page 146](#)
- [“Sr_SetInsertedTablePartElementName\(\)” on page 209](#)
- [“Sw_SetTableScanOrder\(\)” on page 374](#)

SwEventType

FrameMaker exports a FrameMaker document by creating events and conversion objects for each element it encounters in the document. It also generates events to signal the start or end of exporting, or to signal changes in the status of the markup document receiving the exported FrameMaker document. The types of events that can be created on export are enumerated in `SwEventType`.

Event	Meaning
SW_EVT_BEGIN_WRITER	Start of export processing. An import/export client can respond to this event by doing any necessary setup work, such as processing based on settings in the SGML declaration, before the document itself is processed.
SW_EVT_END_WRITER	End of export processing. An import/export client can respond to this event by doing any desired cleanup after all of a document's events are processed.
SW_EVT_BEGIN_BOOK	Start of export from a FrameMaker book.
SW_EVT_END_BOOK	End of export from a FrameMaker book.
SW_EVT_BEGIN_BOOK_COMP	Start of export from a FrameMaker book component (a document that is part of a book). This event occurs instead of <code>SW_EVT_BEGIN_DOC</code> for book components.
SW_EVT_END_BOOK_COMP	End of export from a FrameMaker document that is part of a book. This event occurs instead of <code>SW_EVT_END_DOC</code> for book components.
SW_EVT_BEGIN_DOC	Start of export from a FrameMaker document that is <i>not</i> part of a book.
SW_EVT_END_DOC	End of export from a FrameMaker document that is <i>not</i> part of a book.
SW_EVT_BEGIN_ELEM	Start of a FrameMaker element.
SW_EVT_END_ELEM	End of a FrameMaker element.
SW_EVT_BEGIN_FOOTNOTE	Start of a FrameMaker element representing a footnote.
SW_EVT_END_FOOTNOTE	End of a FrameMaker element representing a footnote.
SW_EVT_BEGIN_TABLE	Start of a FrameMaker element representing a table.
SW_EVT_END_TABLE	End of a FrameMaker element representing a table.
SW_EVT_BEGIN_TABLE_TITLE	Start of a FrameMaker element representing a table title.
SW_EVT_END_TABLE_TITLE	End of a FrameMaker element representing a table title.

Event	Meaning
SW_EVT_BEGIN_TABLE_HEADING	Start of a FrameMaker element representing a table heading.
SW_EVT_END_TABLE_HEADING	End of a FrameMaker element representing a table heading.
SW_EVT_BEGIN_TABLE_BODY	Start of a FrameMaker element representing a table body.
SW_EVT_END_TABLE_BODY	End of a FrameMaker element representing a table body.
SW_EVT_BEGIN_TABLE_FOOTING	Start of a FrameMaker element representing a table footing.
SW_EVT_END_TABLE_FOOTING	End of a FrameMaker element representing a table footing.
SW_EVT_BEGIN_TABLE_ROW	Start of a row in a FrameMaker table.
SW_EVT_END_TABLE_ROW	End of a row in a FrameMaker table.
SW_EVT_BEGIN_TABLE_CELL	Start of a cell in a FrameMaker table.
SW_EVT_END_TABLE_CELL	End of a cell in a FrameMaker table.
SW_EVT_BEGIN_COLSPEC	Start of a <i>colspec</i> in a FrameMaker <i>CALS</i> table.
SW_EVT_END_COLSPEC	End of a <i>colspec</i> in a FrameMaker <i>CALS</i> table.
SW_EVT_BEGIN RUBI_GROUP	Start of a Rubi group in a FrameMaker document.
SW_EVT_END RUBI_GROUP	End of a Rubi group in a FrameMaker document.
SW_EVT_BEGIN RUBI	Start of Rubi text in a FrameMaker document.
SW_EVT_END RUBI	End of Rubi text in a FrameMaker document.
SW_EVT_VARIABLE	Signals a FrameMaker variable element.
SW_EVT_MARKER	Signals a FrameMaker marker element.
SW_EVT_XREF	Signals a FrameMaker cross-reference element.
SW_EVT_GRAPHIC	Signals a FrameMaker graphic element.
SW_EVT_EQUATION	Signals a FrameMaker equation element.
SW_EVT_TEXT	Signals a sequence of text characters in a single font that FrameMaker proposes writing as markup <i>PCDATA</i> , <i>CDATA</i> , or content of a general text entity.
SW_EVT_SPECIAL_CHAR	Signals a character that FrameMaker proposes writing as <i>SDATA</i>
SW_EVT_TEXT_INSET	Signals a FrameMaker text inset element.
SW_EVT_REF_ELEM	Signals a FrameMaker reference element
SW_EVT_EOL	End of a FrameMaker text line that is not the end of a paragraph.
SW_EVT_EOP	End of a FrameMaker paragraph.

Event	Meaning
SW_EVT_CONDITION_CHANGE	Signals a FrameMaker condition change

SwEventTypeT is a data field in the export event structure SwEventT. SwEventT is passed as an argument to Sw_Convert() and Sw_EventHandler(). SwEventT is returned by Sw_GetAssociatedEvent().

See also

- [“Sw_Convert\(\)” on page 283](#)
- [“Sw_EventHandler\(\)” on page 286](#)
- [“Sw_GetAssociatedEvent\(\)” on page 288](#)
- [“SwEventT” on page 425](#)
- [“SwConvObjT” on page 424](#)
- [“SwObjTypeT” on page 407](#)

SwLocationT

When FrameMaker proposes to insert an object in a markup document, it also proposes a general insertion location for the object. General insertion locations are enumerated in SwLocationT.

Value	Meaning
SW_LOC_DTD_SUBSET	Insert into the DTD
SW_LOC_INSTANCE	Insert into the markup document instance

SwLocationT is passed as an argument to Sw_WriteAttrSpec(), Sw_WriteDelimiter(), Sw_WriteReservedName(), and Sw_WriteString().

See also

- [“Sw_WriteAttrSpec\(\)” on page 376](#)
- [“Sw_WriteDelimiter\(\)” on page 377](#)
- [“Sw_WriteReservedName\(\)” on page 379](#)
- [“Sw_WriteString\(\)” on page 380](#)

SwObjTypeT

FrameMaker exports a document by creating events and conversion objects for each element and attribute in the FrameMaker document. The types of export conversion objects that FrameMaker generates are enumerated in SwObjTypeT.

Value	Meaning
SW_OBJ_SESSION	Current markup export session
SW_OBJ_BOOK	FrameMaker book
SW_OBJ_BOOK_COMP	FrameMaker book component
SW_OBJ_DOC	FrameMaker document
SW_OBJ_ELEM	FrameMaker element
SW_OBJ_TABLE	FrameMaker table
SW_OBJ_TABLE_TITLE	Table title
SW_OBJ_TABLE_HEADING	Table heading
SW_OBJ_TABLE_BODY	Table body
SW_OBJ_TABLE_FOOTING	Table footing
SW_OBJ_TABLE_ROW	Table row
SW_OBJ_TABLE_CELL	Table cell
SW_OBJ_COLSPEC	CALS table <i>colspec</i>
SW_OBJ_GRAPHIC	A graphic
SW_OBJ_EQUATION	FrameMaker equation
SW_OBJ_VARIABLE	FrameMaker variable
SW_OBJ_MARKER	FrameMaker marker
SW_OBJ_XREF	FrameMaker cross-reference
SW_OBJ_FOOTNOTE	FrameMaker footnote
SW_OBJ_TEXT	FrameMaker text
SW_OBJ_ENTITY	FrameMaker entity
SW_OBJ_PI	FrameMaker <i>PI</i>
SW_OBJ_RUBI_GROUP	FrameMaker Rubi group
SW_OBJ_RUBI	FrameMaker Rubi text
SW_OBJ_RE	Record end in markup

SwObjTypeT is a data field in an export conversion object structure pointed to by SwConvObjT. An export conversion object's internal structure can differ depending on its object type. Import/export client applications should never directly access an export conversion object. Instead, the client should use structure import/export API calls to return conversion object information.

SwObjTypeT is returned by Sw_GetObjType(), which is called to determine the object's type. For an illustration of this usage, see the example for Sw_EventHandler().

See also

- [“Sw_EventHandler\(\)” on page 286](#)
- [“Sw_GetObjType\(\)” on page 317](#)
- [“SwConvObjT” on page 424](#)

SwTextItemTypeT

When FrameMaker proposes to export text to a markup document, that text can be of several different types. Text types are enumerated in SwTextItemTypeT.

Value	Meaning
SW_TEXT_STRING	Text string
SW_TEXT_NUMERIC_CHAR_REF	Numeric character reference
SW_TEXT_NAMED_CHAR_REF	Named character reference
SW_TEXT_ENT_REF	Entity reference

SwTextItemTypeT is a data field in the text item structure SwTextItemT. SwTextItemT, in turn, is a data field in SwTextItemsT. SwTextItemsT is returned by Sw_GetStructuredText() and is passed as an argument to Sw_SetStructuredText().

See also

- [“Sw_GetStructuredText\(\)” on page 332](#)
- [“Sw_SetStructuredText\(\)” on page 373](#)
- [“SwTextItemT” on page 426](#)
- [“SwTextItemsT” on page 426](#)

Data structures

The following sections describe data structures used by the structure import/export API functions. Each section cross-references the functions that expect these data structures as arguments or that return them as values.

FilePathT

Defines a platform-independent representation of a platform-specific filepath. Unlike most of the other data structures documented below, the internal structure of `FilePathT` is not described. This is because structure import/export API clients should not directly manipulate any fields in `FilePathT`. Fields are not guaranteed to be the same in different versions of the FrameMaker FDK.

To use `FilePathT` in an application, declare a pointer for the structure, and pass the pointer or address of the structure to functions that use `FilePathT` as an argument.

The FDK API contains many functions for translating filepaths to and from platform-specific formats to the generic `FilePathT` format for use in an import/export client. For more information on filepaths, see Chapter 16, "Making I/O and Memory Calls Portable," in the *FDK Programmer's Guide*.

See also

- [“Structured_GetLoc\(\)” on page 75](#)
- [“Sr_GetBookCompFilePath\(\)” on page 116](#)
- [“Sr_GetBookFilePath\(\)” on page 117](#)
- [“Sr_GetExtEntityFilePath\(\)” on page 134](#)
- [“Sr_GetTextInsetFilePath\(\)” on page 174](#)
- [“Sr_SetBookCompFilePath\(\)” on page 189](#)
- [“Sr_SetTextInsetFilePath\(\)” on page 228](#)
- [“Srw_GetExportDtdFilePath\(\)” on page 264](#)
- [“Srw_GetExportSchemaFilePath\(\)” on page 265](#)
- [“Srw_GetImportTemplateFilePath\(\)” on page 266](#)
- [“Srw_GetRulesDocFilePath\(\)” on page 268](#)
- [“Srw_GetStructuredDeclarationFilePath\(\)” on page 270](#)
- [“Srw_GetStructuredDocFilePath\(\)” on page 271](#)
- [“Sw_GetBookCompEntityFilePath\(\)” on page 295](#)
- [“Sw_GetExportFilePath\(\)” on page 309](#)
- [“Sw_SetBookCompEntityFilePath\(\)” on page 350](#)

- [“Sw_SetExportFilePath\(\)” on page 359](#)

F_AttributeT

Describes an individual attribute associated with an import conversion object.

```
typedef struct (  
    StringT name;  
    F_StringsT values;  
    UByteT valflags; /* validation error flag */  
    UByteT allow; /* allow validation error as special case */  
} F_AttributeT;
```

`F_AttributeT.values` is an array of strings containing the attribute values generated by `FrameMaker` on import.

See also

- [“Sr_GetAttrVal\(\)” on page 111](#)
- [“Sr_SetAttrVal\(\)” on page 184](#)
- “`F_ApiDeallocateAttribute()`” in the *FDK Programmer’s Reference*
- [“F_AttributesT” on page 410](#)

F_AttributesT

Provides a list of the attributes associated with an import conversion object.

```
typedef struct {  
    UIntT len; /* number of attributes */  
    F_AttributeT *val; /* attributes array pointer */  
} F_AttributesT;
```

See also

- [“Sr_GetAttrVals\(\)” on page 114](#)
- [“Sr_SetAttrVals\(\)” on page 186](#)
- “`F_ApiDeallocateAttributes()`” in the *FDK Programmer’s Reference*
- [“F_AttributesT” on page 410](#)

StructuredAttrDefT

Deprecated: SgmlAttrDefT

Describes a single markup attribute definition for an element.

```
typedef struct {
    StringT attrName;
    StructuredAttrDeclaredValueT declVal;
    StructuredAttrDefaultValueT defValType;
    StringListT tokenGroup;
    StringListT defVal;
} StructuredAttrDefT;
```

tokenGroup defines the possible values of notation attributes, as well as attributes with declared values that are name-token groups.

defVal represents the default value. Although it is a list of strings, the list contains only one item unless the attribute is a token list. In that case, each token in the default value is a separate item in this list.

See also

- [“StructuredAttrDeclaredValueT” on page 384](#)
- [“StructuredAttrDefaultValueT” on page 385](#)

StructuredAttrDefsT

Deprecated: SgmlAttrDefsT

Provides a list of attribute definitions for an element.

```
typedef struct {
    UIntT len; /* number of attribute definitions */
    StructuredAttrDefT *val; /* attribute defs array pointer */
} StructuredAttrDefsT;
```

StructuredAttrDefsT is a data field in the element definition structure StructuredElementDefT, and in the notation definition structure StructuredNotationDefT.

See also

- [“StructuredAttrDefT” on page 411](#)
- [“StructuredElementDefT” on page 413](#)
- [“StructuredNotationDefT” on page 415](#)

StructuredAttrValT

Deprecated: SgmlAttrValT

Describes an attribute name/attribute-value pair for a markup element start-tag.

```
typedef struct {
    StringT sgmlAttrName;
    StringListT sgmlAttrVal;
    IntT sgmlAttrFlags;
} StructuredAttrValT;
```

`sgmlAttrVal` is a list of strings. Each token in a tokenized attribute is a separate string. Values that are not tokenized appear as the single string in a one-string list.

`sgmlAttrFlags` is a set of flags containing information about the attribute definition. The flags are:

- 0x0001 STRUCTURED_ATTR_VAL_SPECIFIED
- 0x0002 STRUCTURED_ATTR_IS_CDATA
- 0x0004 STRUCTURED_ATTR_IS_ID
- 0x0008 STRUCTURED_ATTR_IS_FIXED
- 0x0010 STRUCTURED_ATTR_IS_IDREF
- 0x0020 STRUCTURED_ATTR_IS_NAME_TOKEN

See also

- [“Structured_DeallocateAttrVal\(\)” on page 55](#)
- [“Structured_IsAttrFixed\(\)” on page 93](#)
- [“Structured_IsAttrNameToken\(\)” on page 94](#)
- [“Structured_IsAttrValSpecified\(\)” on page 95](#)
- [“Structured_IsIdAttr\(\)” on page 96](#)
- [“Sr_GetAttrVal\(\)” on page 111](#)
- [“Sr_SetAttrVal\(\)” on page 184](#)
- [“Sw_GetAttrVal\(\)” on page 290](#)
- [“Sw_SetAttrVal\(\)” on page 345](#)
- [“StructuredAttrValsT” on page 413](#)

StructuredAttrValsT

Deprecated: SgmlAttrValsT

Provides a list of attribute name/attribute-value pairs associated with a start-tag or entity declaration.

```
typedef struct {
    UIntT len; /* number of attributes pairs */
    StructuredAttrValT *val; /* attribute pair array pointer */
    StructuredAttrValsT;
```

StructuredAttrValsT is a data field in the entity definition structure StructuredEntityDefT.

See also

- [“Structured CopyAttrVals\(\)” on page 53](#)
- [“Structured DeallocateAttrVals\(\)” on page 56](#)
- [“Sr GetAttrVals\(\)” on page 114](#)
- [“Sr SetAttrVals\(\)” on page 186](#)
- [“Sw GetAttrVals\(\)” on page 293](#)
- [“Sw SetAttrVals\(\)” on page 347](#)
- [“StructuredAttrValT” on page 412](#)

StructuredElementDefT

Deprecated: SgmlElementDefT

Describes a markup element definition.

```
typedef struct {
    StringT gi;
    StructuredContentTypeT conType;
    UCharT *modelGroup;
    StringListT inclusions, exclusions;
    StructuredAttrDefsT attrs;
    IntT tagOmission;
    IntT reserved;
} StructuredElementDefT;
```

modelGroup is an array in which most characters represent tokens in the XML or SGML model group defined by StructuredContentTokenT.

tagOmission is a flag indicating whether or not start-tags or end-tags can be omitted from the element definition.

The `reserved` field is for internal use. Do not use it.

See also

- [“EndTagOmissible\(\)” on page 50](#)
- [“Structured_GetElementDef\(\)” on page 62](#)
- [“StartTagOmissible\(\)” on page 278](#)
- [“StructuredContentTokenT” on page 385](#)
- [“StructuredContentTypeT” on page 386](#)
- [“StructuredAttrDefsT” on page 411](#)
- [“StructuredEntityDefT” on page 414](#)

StructuredEntityDefT

Deprecated: *SgmlEntityDefT*

Describes a structure containing all information from an entity declaration.

```
typedef struct {
    StringT ename;
    StringT etext;
    StructuredEntityScopeT escope;
    StructuredEntityTypeT etype;
    BoolT external;
    StringT pubid;
    StringT sysid;
    StringT nname;
    StructuredAttrValsT dataAttrVals;
    FilePathT *fp;
} StructuredEntityDefT;
```

If `escope` is `STRUCTURED_ES_PARAMETER`, `ename` is the entity name without the initial *PERO* delimiter.

`etext` is replaceable parameter data. It comes from any parameter literal in the entity declaration subsequent to the replacement of parameter entity references or character references within it.

See also

- [“Structured_GetDefaultEntityDef\(\)” on page 58](#)
- [“Structured_GetEntityDef\(\)” on page 63](#)
- [“StructuredEntityScopeT” on page 388](#)
- [“StructuredEntityTypeT” on page 389](#)
- [“StructuredAttrValsT” on page 413](#)

StructuredNotationDefT

Deprecated: SgmlNotationDefT

A structure for a notation declaration.

```
typedef struct {
    StringT nname;
    StringT pubid;
    StringT sysid;
    StructuredAttrDefsT dataAttrs;
} StructuredNotationDefT;
```

See also

- [“Structured_GetNotationDef\(\)” on page 82](#)
- [“StructuredAttrDefsT” on page 411](#)
- [“StructuredEntityDefT” on page 414](#)

SrConvObjT

Provides a pointer to a data structure for an import conversion object associated with an event.

```
typedef PtrT SrConvObjT;
```

Unlike most of the other data structures documented in this section, the internal structure of an import conversion object is not described. This is because structure import/export API clients should not directly manipulate conversion objects.

To reference an import conversion object, declare a pointer of type `SrConvObjT` to use as an argument or return type with `FrameMaker` API calls.

All possible types of import conversion objects are listed in the `SrObjTypeT` data type enumeration.

See also

- [“SrObjTypeT” on page 395](#)
- [“SwConvObjT” on page 424](#)

SrEventT

Describes a FrameMaker import event.

```
typedef struct {
    SrEventTypeT evtype;
    union {
        SrTagT tag;
        StringT cdata;
        StringT entname;
        StringT pi;
    } u;
} SrEventT;
```

An import event is always paired with a conversion object of type `SrConvObjT`.

See also

- [“Sr_Convert\(\)” on page 104](#)
- [“Sr_EventHandler\(\)” on page 107](#)
- [“Sr_GetAssociatedEvent\(\)” on page 109](#)
- [“SrEventTypeT” on page 393](#)
- [“SrTagT” on page 417](#)
- [“SrConvObjT” on page 415](#)

SrInsertLocT

Describes a structure indicating where in a FrameMaker document a proposed import conversion object should be inserted.

```
typedef struct {
    SrLocationT pos;
    union {
        F_ObjHandleT flowId;
        F_ObjHandleT mkrId;
        F_ObjHandleT tiId;
        F_ElementLocT elemLoc;
    } u;
} SrInsertLocT;
```

`pos` indicates how to interpret the `u` union field of the location structure. It must be set to one of the following values within any given location structure:

- `SR_LOC_FLOW`: `u` contains a `flowId` that indicates the flow into which the first element should be inserted.
- `SR_LOC_BOOK`: `u` is ignored, and the element is inserted into the current book.

- `SR_LOC_ELEMENT`: `u` contains an `elemloc` that indicates the location in the document where the new text, element, or object should be inserted.
- `SR_LOC_MARKER_TEXT`: `u` contains a `mkrId` that indicates the FrameMaker marker into which text should be inserted.
- `SR_LOC_TEXT_INSET`: `u` contains a `tiId` that indicates the FrameMaker text inset into which text should be inserted.

See also

- [“Sr_GetFmObjId\(\)” on page 141](#)
- [“Sr_GetInsertLoc\(\)” on page 147](#)
- [“Sr_SetInsertLoc\(\)” on page 210](#)
- [“SrInsertLocT” on page 416](#)

SrSessionPropsT

Indicates status information about importing to an FrameMaker document.

```
typedef struct {
    BoolT isBatchMode;
    StringT tableRulingStyle;
    BoolT overwriteFiles;
} SrSessionPropsT;
```

`SrSessionPropsT` is returned by `Sr_GetSessionProps()`. It is passed as an argument to `Sr_DeallocateSessionProps()` and `Sr_SetSessionProps()`.

See also

- [“Sr_DeallocateSessionProps\(\)” on page 105](#)
- [“Sr_GetSessionProps\(\)” on page 167](#)
- [“Sr_SetSessionProps\(\)” on page 222](#)
- [“SwSessionPropsT” on page 425](#)

SrTagT

Contains information extracted from a markup element tag.

```
typedef struct {
    StringT gi;
    StructuredAttrValsT sgmlAttrVals;
} SrTagT;
```

`gi` is the generic identifier for the element.

`sgmlAttrVals` is a list of attribute values for the element.

See also

- [“StructuredAttrValsT” on page 413](#)

SRW_errno

Provides a global error variable set by many structure import/export API functions that do not directly return error codes. `SRW_errno` is declared in `fm_base.h`. It indicates whether the last operation was successful or if a particular error occurred. If an error occurs, `SRW_errno` is set to one of the values defined by the enumerated data type `SrwErrorT`.

See also

- [“SrwErrorT” on page 396](#)

SrwColSpecT

Describes a *CALS* table *colspec*.

```
typedef struct {
    IntT colnum;
    StringT colname;
    StringT colwidth;
    StringT alignType;
    StringT alignChar;
    StringT alignOffset;
    BoolT colsep;
    BoolT rowsep;
    UIntT valueSet;
} SrwColSpecT;
```

`valueSet` indicates which other data fields in `SrwColSpecT` contain real information. It can be set to any combination of the following flags:

- `SRW_COLSPEC_COLNUM`
- `SRW_COLSPEC_COLNAME`
- `SRW_COLSPEC_COLWIDTH`
- `SRW_COLSPEC_ALIGN`
- `SRW_COLSPEC_CHAROFF`
- `SRW_COLSPEC_CHAR`
- `SRW_COLSPEC_COLSEP`
- `SRW_COLSPEC_ROWSEP`

For example, to indicate that `colnum`, `colname`, and `alignOffset` contain values that should be used when interpreting a particular `SrwColSpecT` structure (here `MyColSpec`) you would set `valueSet` using the following code:

```
. . .
MyColSpec.valueSet = SRW_COLSPEC_COLNUM | SRW_COLSPEC_COLNAME |
                    SRW_COLSPEC_CHAROFF;
. . .
```

`SrwColSpecT` is a data field in `SrwColSpecsT`. It is returned by `Sr_GetCurColSpecByName()`, `Sr_GetCurColSpecByColNum()`, `Srw_GetColSpecByName()`, `Srw_GetColSpecByColNum()`, and `Srw_CopyColSpec()`. `SrwColSpecT` is passed as an argument to `Srw_SetColSpec()`, `Srw_CopyColSpec()`, and `Srw_DeallocateColSpec()`.

See also

- [“Sr_GetColSpecs\(\)” on page 124](#)
- [“Sr_GetCurColSpecByColNum\(\)” on page 126](#)
- [“Sr_GetCurColSpecByName\(\)” on page 127](#)
- [“Srw_GetColSpecByColNum\(\)” on page 261](#)
- [“Srw_GetColSpecByName\(\)” on page 262](#)
- [“Srw_CopyColSpecs\(\)” on page 240](#)
- [“Srw_SetColSpec\(\)” on page 275](#)
- [“Srw_DeallocateColSpec\(\)” on page 250](#)
- [“SrwColSpecsT” on page 419](#)
- [“SrwSpanSpecT” on page 422](#)

SrwColSpecsT

Lists *colspecs* associated with a table or table-part (heading, body, footing) import conversion object.

```
typedef struct {
    IntT len; /* Number of colspecs in array. */
    SrwColSpecT *val; /* Pointer to array of colspecs. */
} SrwColSpecsT;
```

`SrwColSpecsT` is returned by `Sr_GetColSpecs()`, `Sw_GetColSpecs()`, and `Srw_CopyColSpecs()`. It is passed as an argument to `Sr_SetColSpecs()`, `Sw_SetColSpecs()`, `Srw_CopyColSpecs()`, and `Srw_DeallocateColSpecs()`.

See also

- [“Sr_GetColSpecs\(\)” on page 124](#)

- [“Sr_SetColSpecs\(\)” on page 195](#)
- [“Srw_CopyColSpecs\(\)” on page 240](#)
- [“Srw_DeallocateColSpecs\(\)” on page 251](#)
- [“Sw_GetColSpecs\(\)” on page 300](#)
- [“Sw_SetColSpecs\(\)” on page 354](#)
- [“SrwColSpecT” on page 418](#)
- [“SrwSpanSpecsT” on page 423](#)

SrwLogMessageLocationT

Indicates the document and location within a document to associate with a message to be written to a log file.

```
typedef struct {
    SrwLogDocT docIdentType;
    union {
        FilePathT *fp;
        F_ObjHandleT docId;
    } doc_u;
    SrwLogLocT locType;
    union {
        F_ObjHandleT targetId;
        IntT line;
    } loc_u;
} SrwLogMessageLocationT
```

`targetId` can be specified only if `docIdentType` is `SRW_LOGD_ID`.

See also

- [“Srw_LogMessage\(\)” on page 273](#)
- [“SrwLogDocT” on page 402](#)
- [“SrwLogLocT” on page 403](#)

SrwPropValT

Describes a FrameMaker property for a markup element or FrameMaker object.

```
typedef struct {
    SrwFmPropertyT prop;
    StringT value;
} SrwPropValT;
```

`prop` identifies the type of property, and `value` is the value (for example, the number of columns).

`SrwPropValT` is a data field in `SrwPropValsT`. It is returned by `Sr_GetPropVal()` and the value is returned by `Sw_GetPropVal()`. It is passed as an argument to `Sr_SetPropVal()` and `Srw_DeallocatePropVal()`.

See also

- [“Sr_GetPropVal\(\)” on page 159](#)
- [“Sr_SetPropVal\(\)” on page 217](#)
- [“Srw_DeallocatePropVal\(\)” on page 252](#)
- [“Sw_GetPropVal\(\)” on page 324](#)
- [“SrwFmPropertyT” on page 397](#)
- [“SrwPropValsT” on page 421](#)

SrwPropValsT

Provides a list of all properties for a single markup element or FrameMaker object.

```
typedef struct {
    UIntT len; /* Number of properties in array. */
    SrwPropValT *val; /* Pointer to array of values. */
} SrwPropValsT;
```

`SrwPropValsT` is returned by `Sr_GetPropVals()` and `Sw_GetPropVals()`. It is passed as an argument to `Sr_SetPropVals()` and `Srw_DeallocatePropVals()`.

See also

- [“Sr_GetPropVals\(\)” on page 162](#)
- [“Sr_SetPropVals\(\)” on page 219](#)
- [“Srw_DeallocatePropVals\(\)” on page 253](#)
- [“Sw_GetPropVals\(\)” on page 326](#)
- [“SrwPropValT” on page 420](#)

SrwSpanSpecT

Describes a CALS table *spanspec*.

```
typedef struct {
    StringT spanname;
    StringT namest;
    StringT nameend;
    StringT alignType;
    StringT alignChar;
    StringT alignOffset;
    BoolT colsep;
    BoolT rowsep;
    UIntT valueSet;
} SrwSpanSpecT;
```

`valueSet` indicates which other data fields in `SrwSpanSpecT` contain real information. It can be set to any combination of the following flags:

- `SRW_SPANSPEC_SPANNAME`
- `SRW_SPANSPEC_NAMEST`
- `SRW_SPANSPEC_NAMEEND`
- `SRW_SPANSPEC_ALIGN`
- `SRW_SPANSPEC_CHAROFF`
- `SRW_SPANSPEC_CHAR`
- `SRW_SPANSPEC_COLSEP`
- `SRW_SPANSPEC_ROWSEP`

For example, to indicate that `spanname`, `namest`, and `nameend` contain values that should be used when interpreting a particular `SrwSpanSpecT` structure (here `MySpanSpec`) structure, you would set `valueSet` using the following code:

```
. . .
MySpanSpec.valueSet = SRW_SPANSPEC_SPANNAME |
    SRW_SPANSPEC_NAMEST | SRW_SPANSPEC_NAMEEND;
. . .
```

`SrwSpanSpecT` is a data field in `SrwSpanSpecsT`. It is returned by `Sr_GetCurSpanSpecByName()`, `Srw_GetSpanSpecByName()`, and `Srw_CopySpanSpec()`. `SrwSpanSpecT` is passed as an argument to `Srw_SetSpanSpec()`, `Srw_CopySpanSpec()`, and `Srw_DeallocateSpanSpec()`.

See also

- [“`Sr_GetCurSpanSpecByName\(\)`” on page 131](#)
- [“`Srw_GetSpanSpecByName\(\)`” on page 272](#)

- [“Srw_CopySpanSpec\(\)” on page 245](#)
- [“Srw_DeallocateColSpec\(\)” on page 250](#)
- [“Srw_SetSpanSpec\(\)” on page 276](#)
- [“SrwSpanSpecsT” on page 423](#)
- [“SrwColSpecT” on page 418](#)

SrwSpanSpecsT

Provides a list of *spanspecs* associated with a table or table-part (heading, body, footing) conversion object.

```
typedef struct {
    IntT len; /* Number of spanspecs in array. */
    SrwSpanSpecT *val; /* Pointer to array of spanspecs. */
} SrwSpanSpecsT;
```

SrwSpanSpecsT is returned by *Sr_GetSpanSpecs()* and *Srw_CopySpanSpecs()*. It is passed as an argument to *Sr_SetSpanSpecs()*, *Srw_CopySpanSpecs()*, and *Srw_DeallocateSpanSpecs()*.

See also

- [“Sr_GetSpanSpecs\(\)” on page 169](#)
- [“Sr_SetSpanSpecs\(\)” on page 224](#)
- [“Srw_CopySpanSpecs\(\)” on page 246](#)
- [“Srw_DeallocateSpanSpecs\(\)” on page 255](#)
- [“SrwSpanSpecT” on page 422](#)
- [“SrwColSpecsT” on page 419](#)

SrwStraddleT

Describes a structure representing a running straddle in a non-CALS table conversion object.

```
typedef struct {
    F_ObjHandleT cellId;
    StringT straddleName;
} SrwStraddleT;
```

SrwStraddleT is a data field in *SrwStraddlesT*. It is returned by *Srw_CopyStraddle()*, and is passed as an argument to *Srw_SetStraddle()*, *Srw_CopyStraddle()*, and *Srw_DeallocateStraddle()*.

See also

- [“Srw_CopyStraddle\(\)” on page 247](#)

- [“Srw_DeallocateStraddle\(\)” on page 256](#)
- [“Srw_SetStraddle\(\)” on page 277](#)
- [“SrwStraddlesT” on page 424](#)

SrwStraddlesT

Provides a list of running straddles in a non-CALS table conversion object.

```
typedef struct {
    IntT len; /* Number of straddles in array. */
    SrwStraddleT *val; /* Pointer to array of straddles. */
} SrwStraddlesT;
```

`SrwStraddlesT` is returned by `Sr_GetStraddles()`, and `Srw_CopyStraddles()`. It is passed as an argument to `Sr_SetStraddles()`, `Srw_CopyStraddles()`, `Srw_DeallocateStraddles()`, and `Srw_DeleteStraddlesByName()`.

See also

- [“Sr_GetStraddles\(\)” on page 170](#)
- [“Sr_SetStraddles\(\)” on page 225](#)
- [“Srw_CopyStraddles\(\)” on page 248](#)
- [“Srw_DeallocateStraddles\(\)” on page 257](#)
- [“Srw_DeleteStraddlesByName\(\)” on page 258](#)
- [“SrwStraddleT” on page 423](#)

SwConvObjT

Describes a pointer to a data structure for an export conversion object associated with an event.

```
typedef PtrT SwConvObjT;
```

Unlike most of the other data structures documented in this section, the internal structure of an export conversion object is not described. This is because structure import/export API clients should not directly manipulate conversion objects.

To reference an export conversion object, declare a pointer of type `SwConvObjT` to use as an argument or return type with `FrameMaker` API calls.

All possible types of export conversion objects are listed in the `SwObjTypeT` data type enumeration.

See also

- [“SwConvObjT” on page 424](#)
- [“SrConvObjT” on page 415](#)

SwEventT

Describes a FrameMaker structure for an export event.

```
typedef struct {
    SwEventTypeT evtype;
    F_TextLocT txtloc;
    F_ObjHandleT fm_elemid;
    F_AttributesT fm_attrs;
    F_ObjHandleT fm_objid;
    StringT text;
    UCharT charcode;
    StringT chartag;
} SwEventT;
```

An export event is always paired with a conversion object of type `SwConvObjT`.

`fm_elemid` is always set. It is the ID of the FrameMaker element.

`fm_attrs` is set only for container elements.

`fm_objid` is the ID of the object in an element or in nonelement variables and markers.

`text` is only set for text strings.

`charcode` and `chartag` are only set for special characters.

See also

- [“Sw_Convert\(\)” on page 283](#)
- [“Sw_EventHandler\(\)” on page 286](#)
- [“Sw_GetAssociatedEvent\(\)” on page 288](#)
- [“SrEventTypeT” on page 393](#)

SwSessionPropsT

Indicates status information about exporting to a markup document.

```
typedef struct {
    BoolT includeDtd;
    BoolT includeSgmlDecl;
    StringT doctypeSysId;
    StringT doctypePubId;
    BoolT overwriteFiles;
} SwSessionPropsT;
```

`SwSessionPropsT` is returned by `Sw_GetSessionProps()`. It is passed as an argument to `Sw_DeallocateSessionProps()` and `Sw_SetSessionProps()`.

`overwriteFiles` is true if batch can overwrite files of the same name in the target directory.

See also

- [“Sw_DeallocateSessionProps\(\)” on page 284](#)
- [“Sw_GetSessionProps\(\)” on page 329](#)
- [“Sw_SetSessionProps\(\)” on page 370](#)
- [“SrSessionPropsT” on page 417](#)

SwTextItemT

Provides a structure representing a single sequence of FrameMaker characters to be represented in markup as data characters, a character reference, or an entity reference.

```
typedef struct {
    SwTextItemTypeT itemType;
    union {
        StringT text;
        UCharT charNum;
        StringT charName;
        StringT entName;
    } u;
} SwTextItemT;
```

`SwTextItemT` is a data field in `SwTextItemsT`.

See also

- [“SwTextItemsT” on page 426](#)

SwTextItemsT

Describes a list of text items associated with an export conversion object, each of which represents markup data characters, a character reference, or an entity reference.

```
typedef struct {
    UIntT len;
    SwTextItemT *val;
} SwTextItemsT;
```

`SwTextItemsT` is returned by `Sw_GetStructuredText()` and is passed as an argument to `Sw_SetStructuredText()`.

See also

- [“Sw_GetStructuredText\(\)” on page 332](#)
- [“Sw_SetStructuredText\(\)” on page 373](#)
- [“SwTextItemT” on page 426](#)

Index

To go to a page, click on a page number below.

A

API

- alphabetic list of import/export
 - functions 49-380
- classes of import/export functions 49
- functions
 - DTD and SGML declaration 50-96, 278
 - export 281-380
 - import 99-237
 - import and export 239-277
 - see also individual function names*
- standard FDK *see the FDK Programmer's Reference*
- structured markup export conversion
 - routine 17
- structured markup import conversion
 - routine 17
- APPINFO parameter 57
- application definition file, *see* structapps.fm
- attribute lists
 - copying 51, 53
 - freeing 56
 - getting on import 114, 293
 - graphic and equation for export
 - getting 310
 - setting 360
- attributes
 - copying 51, 53
 - fixed, testing for 91, 93, 97
 - freeing 56
 - getting case-sensitivity of 71

- graphic and equation values for export

- getting 310

- setting 360

- testing for a name group token 94

- testing for ID value 96

- values

- setting on import 184

- testing for specified 95

- writing a data-attribute specification to the

- DTD 376

- writing a start-tag specification to a document

- instance 376

- audience, intended 9

B

book components

- filepath

- entity for export, getting 295

- entity for export, setting 350

- getting import 116

- setting import 189

PI

- getting export 296

- setting export 351

book files

- filepath

- getting import 117

- setting import 190

PI

- getting export 297

- setting export 352

-
- book ID
 - getting for import 119, 132, 135, 138, 141, 192, 304
 - getting the main FrameMaker 267
 - building
 - import/export clients 28
 - initial conversion objects 10
 - C**
 - CALS
 - colspec *see* colspecs
 - table *see* tables
 - canceling import
 - all files 102
 - cells
 - marking as used 226
 - testing for forcing of new rows 172
 - testing for used 103
 - character format tags
 - getting 120
 - setting 194
 - child conversion objects
 - getting GI of siblings 154
 - getting import 122
 - clients
 - building 28
 - compiling 28
 - DocBook Starter Kit described 12
 - event handlers in 23
 - header files for 22
 - registering 28
 - structured markup
 - column-major tables and 12
 - constants in 22
 - conversion objects and 13
 - events and 13
 - parsing of document 10
 - processing instructions and 12
 - tasks requiring 12
 - colspecs
 - copying 239
 - copying a list of 240
 - freeing 250
 - getting by column name from a list 262
 - getting by column name on import 127
 - getting by column number from a list 261
 - getting by column number on import 126
 - getting for export 300
 - getting imported 124
 - inserting a new colspec in a list 275
 - setting for export 354
 - setting import 195
 - updating an existing colspec in a list 275
 - column-major tables 12
 - compiling import/export clients 28
 - CONCUR 84
 - constants in import/export clients 22
 - conversion objects
 - built and proposed by FrameMaker 10
 - converting export 283
 - defined 13
 - flow chart of conversion process 16
 - getting by object type 129
 - getting child objects of import 122
 - getting current import 128
 - getting parent of import 153
 - getting type of import 152
 - converting
 - conversion objects on export 283
 - conversion objects on import 104
 - flow chart of FrameMaker and import/export client interaction during 16
-

copying

- attribute lists 51, 53
- attributes 51, 53
- colspec lists 240
- colspecs 239
- spanspec lists 246
- spanspecs 245
- straddle lists 248
- straddles 247

D

- data structures 409-426
- data types 383-408
- data-content notation
 - definition of in DTD 82
 - getting case-sensitivity of names 71
 - getting name of first in DTD 70
 - getting name of next in DTD 80
- DATATAG 84
- deallocating, *see* freeing
- declaration, *see* SGML declaration
- default import/export client, overview of 19
- defining an import/export client in
 - structapps.fm 28
- delimiter string
 - getting 59
 - writing to a document instance 377
 - writing to the DTD 377
- DocBook Starter Kit 12
- document
 - attribute, writing a start-tag specification to the 376
 - delimiter string, writing to 377
 - import/export client parsing of 10
 - writing a reserved name to the 379
 - writing a string to the 380

document ID

- getting the main FrameMaker 267
- setting FrameMaker for import 197

DTD

- attribute, writing a data specification to the 376
- data-content notation definition 82
- delimiter string, writing to 377
- element definitions in 62
- element names
 - getting first in 67
 - getting next in 77
- entity names
 - getting first in 68
 - getting next in 78
- getting DOCTYPE name from 61
- notation names
 - getting first in 70
 - getting next in 80
- writing a reserved name to the 379
- writing a string to the 380

E

elements

- case-insensitivity of names in reference
 - concrete syntax 62
- FrameMaker ID
 - setting 237
- FrameMaker reference
 - setting name of 221
- FrameMaker tag
 - getting 140
 - setting 203
- getting case-sensitivity of 71
- getting definition of from DTD 62
- getting name of first in DTD 67

- getting name of next in DTD 77
- inserted table part
 - getting name of 146
 - setting name of 209
- processing flags
 - getting import 158, 323
- end-tag, notifying FrameMaker about writing out an 337
- end-tag omission 50
- EndTagOmissible() 50
- entities
 - book component, export filepath, getting 295
 - default definition of 58
 - general
 - getting name of first in DTD 68
 - getting name of next in DTD 78
 - testing for specifically named 335
 - getting name of first in DTD 68
 - getting name of next in DTD 78
 - graphic and equation for export
 - getting name of 311
 - getting type of 312
 - setting name of 361
 - setting type of 362
 - name of export
 - getting 306
 - setting 356
 - notifying FrameMaker about writing out a definition for 338
 - parameter
 - getting name of first in DTD 68
 - getting name of next in DTD 78
 - testing for previously used names 336
- equations
 - attribute lists
 - getting export 310

- setting for export 360
- attributes
 - getting for export 310
 - setting for export 360
- entities
 - getting name of for export 311
 - getting type of for export 312
 - setting name of for export 361
 - setting type of for export 362
- file format
 - getting export 307
 - getting import 145
 - setting export 357
 - setting import 206
- notation names
 - getting for export 314
 - setting for export 364
- public identifier
 - getting for export 315
 - setting for export 365
- system identifier
 - getting for export 316
 - setting for export 366
- event handlers
 - example of 17
 - in import/export clients 23
 - Sr_EventHandler() and import event/
 - conversion object pairs 107
 - Sw_EventHandler() and export event/
 - conversion object pairs 259, 286
 - switch statements in 23
- events
 - associated with export conversion objects,
 - getting 288
 - associated with import conversion objects,
 - getting 109

- defined 13
- how FrameMaker builds initial 10
- EXPLICIT 84
- export conversion routine 17
- exporting
 - colspecs
 - getting 300
 - setting 354
 - entity
 - getting name of 306
 - getting name of graphic and equation 311
 - setting name of 356
 - setting name of graphic and equation for 361
- file format for graphics and equations
 - getting 307
 - setting 357
- filepaths
 - getting graphics and equations for 309
 - setting graphics and equations for 359
- graphic and equation entity
 - getting type of 312
 - setting type of 362
- notation names for graphic and equation
 - getting 314
 - setting 364
- overview of structured markup 10
- process described 13

F

- F_AttributesT 410
- F_AttributeT 410
- FDE
 - string, string list, character, I/O, and memory libraries 22
- FDK
 - API 10

- structure import/export API 10
- FDK Programmer's Guide* 22, 49
- FDK Reference Guide* 10
- file format
 - export graphic and equation
 - getting 307
 - setting 357
 - import graphic and equation
 - getting 145
 - setting 206
 - table of processing choices for export 307, 357
- filepaths
 - book component
 - getting for export 295
 - getting import 116
 - setting for export 350
 - setting import 189
 - book file
 - getting import 117
 - setting import 190
- equations
 - getting export 309
 - setting export 359
- external entity, getting 134
- graphics
 - getting export 309
 - setting export 359
- text inset
 - getting 174
 - setting 229
- FilePathT 409
- filter, *see* import filters
- fixed attributes, testing for 91, 93, 97
- flow
 - setting ID for 199

- text insets
 - getting 176
 - setting 231
- fm_sgml.h 49
- fm_struct.h 22
- FmTranslator 16
 - C code for 19
 - overview of 19
- FORMAL 83
- format tags, *see* character format tags 120
- Frame Development Environment 22
- freeing
 - attribute lists 56
 - attributes 56
 - colspecs 250
 - import session properties 105
 - straddles
 - by name 258
- FV_Above 99
- FV_Below 99
- FV_Body 99
- FV_Footing 99
- FV_Heading 99

G

- general entities
 - getting name of first in DTD 68
 - getting name of next in DTD 78
 - notifying FrameMaker about writing out a
 - definition for 338
 - testing for previously used names 336
 - testing for specifically named 335
- generic identifier 67
- GI 67
 - of child conversion object 154
 - of export conversion objects

- setting 372
- global declarations in import/export clients 22
- graphics
 - attribute lists
 - getting export 310
 - setting for export 360
 - attributes
 - getting for export 310
 - setting for export 360
- entities
 - getting name of for export 311
 - getting type of for export 312
 - setting name of for export 361
 - setting type of for export 362
- file format
 - getting export 307
 - getting import 145
 - setting export 357
 - setting import 206
- notation names
 - getting for export 314
 - setting for export 364
- public identifier
 - getting for export 315
 - setting for export 365
- system identifier
 - getting for export 316
 - setting for export 366

H

- header files in import/export clients 22

I

ID

- book
 - getting for import 119, 132, 135, 138, 141,

- 192, 304
- document
 - setting FrameMaker for import 197
- elements
 - setting FrameMaker 237
- flow
 - setting 199
- public for exporting graphics and equations
 - getting 315
 - setting 365
- system for exporting graphics and equations
 - getting 316
 - setting 366
- template document, getting 173
- value, testing an attribute for a 96
- identifier, *see* generic identifier
- IMPLICIT 84
- import conversion API function 17
- import filters
 - graphic and equation
 - getting 145
 - setting 206
- importing
 - book ID
 - getting 119, 132, 135, 138, 141, 192, 304
 - canceling for all files 102
 - child conversion objects, getting 122
 - colspecs
 - getting 124
 - getting by column name 127
 - getting by column number 126
 - setting 195
 - conversion object, getting current 128
 - conversion objects by type, getting 129
 - document ID
 - setting FrameMaker 197

- flow ID
 - setting 199
- FrameMaker element ID
 - setting 237
- insert location
 - getting 147
 - setting 210
- overview of structured markup 10
- process described 13
- processing flags
 - getting 158, 323
- property value
 - setting a single 217
- session properties
 - getting 167, 329
- spanspecs
 - getting 169
 - getting by span name 131
 - setting 224
- special characters
 - getting 137
 - setting 202
- straddles
 - getting 170
 - setting 225
- template document ID, getting 173
- insert location
 - getting import 147
 - setting import 210

L

- LCNMCHAR parameter 73
- LCNMSTRT parameter 74
- line-break information
 - getting import 150
 - setting import 213

N

- name group token, testing an attribute for 94
- NAMECASE ENTITY parameter 65
- NAMECASE GENERAL parameter 71
- notation, DTD definition of data-content 82
- notation names
 - getting case-sensitivity of 71
 - getting first in DTD 70
 - getting graphic and equation for export 314
 - getting next in DTD 80
 - setting graphic and equation for export 364

O

- OMIT_END_TAG 50
- OMIT_START_TAG 278
- OMITTAG 83

P

- page space
 - getting for text insets 177, 179, 232
 - setting for text insets 234
- parameter entities
 - getting name of first in DTD 68
 - getting name of next in DTD 78
- parameter open entity reference, *see* PERO
 - delimiter
- parent conversion objects
 - getting import 153
- parsing
 - as an import/export client task 10
 - read/write rules file 13
- PERO delimiter 64, 69
- PI
 - export book component
 - getting 296
 - setting 351

- export book file
 - getting 297
 - setting 352
- export conversion objects
 - getting 319
 - setting 367
- import/export clients and 12
- prerequisites
 - Structured Application Developer's Guide* 9
 - C and C++ 9
 - FDE 9
 - FDK 9
 - XML and SGML 9
- processing flags
 - import
 - getting 158, 323
- processing instructions, *see* PI 12
- property value
 - setting an import 217
- public identifier
 - for exporting graphics and equations
 - getting 315
 - setting 365

R

- RANK 84
- read/write rules file
 - DTD and 13
 - parsing 13
 - SGML declaration and 13
- reference concrete syntax
 - case-insensitivity of element names 62
- reference elements
 - setting name of FrameMaker 221
- registering import/export clients 28
- reserved names

- writing to a document instance 379
- writing to the DTD 379
- rows
 - marking as used 227
 - testing for use 182
- running straddles, *see* straddles

S

- scan order for tables, setting for export 375
- session properties
 - deallocating import 105
 - getting import 167, 329
- SGML declaration
 - LCNMCHAR parameter 73
 - LCNMSTRT parameter 74
 - read/write rules file and 13
 - UCNMCHAR parameter 89
 - UNCNMSTRT parameter 90
- SHORTREF 83
- SHORTTAG 83
- sibling conversion object, *see* child conversion object
- SIMPLE 84
- spanspecs
 - copying 245
 - copying a list of 246
 - getting by span name on import 131
 - getting import 169
 - inserting a new spanspec in a list 276
 - setting import 224
 - updating an existing spanspec in a list 276
- special characters
 - getting on import 137
 - setting import 202
- Sr_AddTableRows() 99
- Sr_CancelCurBatchFile() 100

- Sr_CancelOperation() 102
- Sr_CellInUse() 103
- Sr_Convert() 17, 104
- Sr_DeallocateSessionProps() 105
- Sr_EventHandler() 107
- Sr_GetAssociatedEvent() 109
- Sr_GetAttrVal() 111
- Sr_GetAttrVals() 114, 293
- Sr_GetBookCompFilePath() 116
- Sr_GetBookFilePath() 117
- Sr_GetBookId() 119
- Sr_GetCharFmt() 120
- Sr_GetChildConvObj() 122
- Sr_GetColSpecs() 124
- Sr_GetCurColSpecByColNum() 126
- Sr_GetCurColSpecByName() 127
- Sr_GetCurConvObj() 128
- Sr_GetCurConvObjOfType() 129
- Sr_GetCurSpanSpecByName() 131
- Sr_GetDocId() 132, 304
- Sr_GetExtEntityFilePath() 134
- Sr_GetFlowId() 135
- Sr_GetFmChar() 137
- Sr_GetFmElemId() 138
- Sr_GetFmElemTag() 140
- Sr_GetFmObjId() 141
- Sr_GetFmText() 143
- Sr_GetImportFileHint() 145
- Sr_GetInsertedTablePartElementName() 146
- Sr_GetInsertLoc() 147
- Sr_GetLineBrkInfo() 150
- Sr_GetObjType() 152
- Sr_GetParentConvObj() 153
- Sr_GetPrevStructuredGi() 154
- Sr_GetPrivateData() 155
- Sr_GetProcessingFlags() 158

Sr_GetPropVal() 159	Sr_SetStraddles() 225
Sr_GetPropVals() 162	Sr_SetTableCellUsed() 226
Sr_GetRefElemTag() 165	Sr_SetTableRowUsed() 227
Sr_GetSessionProps() 167	Sr_SetTextInsetFilePath() 228
Sr_GetSpanSpecs() 169	Sr_SetTextInsetFlowTag() 231
Sr_GetStraddles() 170	Sr_SetTextInsetFormatting() 232
Sr_GetTemplateDocId() 173	Sr_SetTextInsetPageSpace() 234
Sr_GetTextInsetFilePath() 174	Sr_SetVariableName() 235
Sr_GetTextInsetFlowTag() 176	Sr_UseFmElemId() 236
Sr_GetTextInsetFormatting() 177	SrConvObjT 415
Sr_GetTextInsetPageSpace() 179	SrEventT 416
Sr_GetVariableName() 180	SrEventTypeT 393
Sr_RowInUse() 182	SrInsertLocT 416
Sr_SetAttrVal() 184	SrLocationT 394
Sr_SetAttrVals() 186	SrObjTypeT 395
Sr_SetBookCompFilePath() 189	SrSessionPropsT 417
Sr_SetBookFilePath() 190	SrTagT 417
Sr_SetBookId() 192	Srw_CopyColSpec() 239
Sr_SetCharFmt() 194	Srw_CopyColSpecs() 240
Sr_SetColSpecs() 195	Srw_CopySpanSpec() 245
Sr_SetDocId() 197	Srw_CopySpanSpecs() 246
Sr_SetFlowId() 199	Srw_CopyStraddle() 247
Sr_SetFmChar() 202	Srw_CopyStraddles() 248
Sr_SetFmElemTag() 203	Srw_DeallocateColSpec() 250
Sr_SetFmText() 205	Srw_DeallocateColSpecs() 251
Sr_SetImportFileHint() 206	Srw_DeallocatePropVal() 252
Sr_SetInsertedTablePartElementName() 209	Srw_DeallocatePropVals() 253
Sr_SetInsertLoc() 210	Srw_DeallocateSpanSpec() 254
Sr_SetLineBrkInfo() 213	Srw_DeallocateSpanSpecs() 255
Sr_SetPrivateData() 215	Srw_DeallocateStraddle() 256
Sr_SetProcessingFlags() 216	Srw_DeallocateStraddles() 257
Sr_SetPropVal() 217	Srw_DeleteStraddlesByName() 258
Sr_SetPropVals() 219	SRW_errno 418
Sr_SetRefElemTag() 221	Srw_GetColSpecByColNum() 261
Sr_SetSessionProps() 222	Srw_GetColSpecByName() 262
Sr_SetSpanSpecs() 224	Srw_GetExportDtdFilePath() 264, 265

Srw_GetImportTemplateFilePath() 266
 Srw_GetMainDocBookId() 267
 Srw_GetRulesDocFilePath() 268
 Srw_GetSpanSpecByName() 272
 Srw_GetStructuredDeclarationFilePath() 270
 Srw_GetStructuredDocFilePath() 271
 Srw_LogMessage() 273
 Srw_SetColSpec() 275
 Srw_SetSpanSpec() 276
 Srw_SetStraddle() 277
 SrwColSpecsT 419
 SrwColSpecT 418
 SrwErrorT 396
 SrwFmPropertyT 397
 SrwFmPropValT 401
 SrwLogDocT 402
 SrwLogLocT 403
 SrwLogMessageLocationT 420
 SrwPropValsT 421
 SrwPropValT 420
 SrwSpanSpecsT 423
 SrwSpanSpecT 422
 SrwStraddlesT 424
 SrwStraddleT 423
 SrwTablePartTypeT 403
 Starter Kit, DocBook 12
 start-tag, notifying FrameMaker about writing out
 a 339
 start-tag omission 278
 StartTagOmissible() 278
 straddles
 copying 247
 copying a list of 248
 freeing by name 258
 getting on import 170
 inserting a new straddle in a list 277
 setting import 225
 updating an existing straddle in a list 277
 strings
 writing to a document instance 380
 writing to the DTD 380
 structapps.fm 10, 13
 defining a client in 28
 structured markup
 API conversion routine 17
 API, FDK 10
 API functions *see* API
 attribute lists
 copying 51, 53
 freeing 56
 getting on import 114, 293
 graphic and equation values for export,
 getting 310
 graphic and equation values for export,
 setting 360
 attributes
 copying 51, 53
 freeing 56
 getting case-sensitivity of 71
 graphic and equation values for export,
 getting 310
 graphic and equation values for export,
 setting 360
 testing for a name group token 94
 testing for fixed 91, 93, 97
 testing for ID value 96
 testing for specified values 95
 writing a data-attribute specification to the
 DTD 376
 writing a start-tag specification to a document
 instance 376
 clients

- building 28
- column-major tables and 12
- compiling 28
- constants in 22
- conversion objects and 13
- creating 21-??
- defining in structapps.fm 28
- event handlers in 23
- events and 13
- export process described 13
- global declarations in 22
- header files for 22
- import process described 13
- interaction with FrameMaker during
 - conversion 16
- parsing by 10
- processing instructions and 12
- registering 28
- declaration
 - getting APPINFO parameter from 57
 - getting delimiter strings from 59
 - LCNMCHAR parameter 73
 - LCNMSTRT parameter 74
 - NAMECASE ENTITY parameter 65
 - NAMECASE GENERAL parameter 71
 - read/write rules file and 13
 - UCNMCHAR parameter 89
 - UCNMSTRT parameter 90
- delimiter string
 - getting 59
 - writing to a document instance 377
 - writing to the DTD 377
- DocBook Starter Kit client described 12
- document
 - parsing by clients 10
- DTD
 - DOCTYPE name from 61
 - read/write rules file and 13
- elements
 - getting case-sensitivity of names 71
 - getting name of first in DTD 67
 - getting name of next in DTD 77
- end-tag, notifying FrameMaker about writing
 - out an 337
- entities
 - getting name of export 306
 - getting name of first in DTD 68
 - getting name of graphic and equation for
 - export 311
 - getting name of next in DTD 78
 - getting type of graphic and equation for
 - export 312
 - notifying FrameMaker about writing out a
 - definition for 338
 - setting name of export 356
 - setting name of graphic and equation for
 - export 361
 - setting type of graphic and equation for
 - export 362
 - testing for previously used names 336
 - testing for specifically named 335
- exporting process described 10
- generic identifier (GI) 67
- getting default entity definition 58
- getting delimiter string 59
- GI
 - setting export conversion objects 372
- import process described 10
- LCNMCHAR parameter 73
- LCNMSTRT parameter 74
- NAMECASE ENTITY parameter 65
- NAMECASE GENERAL parameter 71

notation names	UCNMCHAR parameter 89
getting case-sensitivity of 71	UCNMSTRT parameter 90
getting graphic and equation for export 314	Structured_ApiCommand() 20
getting name of first in DTD 70	Structured_ApiDialogEvent() 20
getting name of next in DTD 80	Structured_ApiEmergency() 20
setting graphic and equation for export 364	Structured_ApiInitialize() 20
overview of exporting to 10	Structured_ApiNotify() 20
overview of importing from 10	Structured_CopyAttrVals() 51, 53
PERO 64, 69	Structured_DeallocateAttrVal() 55
PI	Structured_DeallocateAttrVals() 56
getting export 319	Structured_GetAppinfo() 57
getting export book component 296	Structured_GetDefaultEntityDef() 58
getting export book file 297	Structured_GetDelimiterString() 59
setting export 367	Structured_GetDocTypeName() 61
setting export book component 351	Structured_GetElementDef() 62
setting export book file 352	Structured_GetEntityDef() 63
public identifier	Structured_GetEntityNamecase() 65
getting graphic and equation for export 315	Structured_GetFirstElementName() 67
setting graphic and equation for export 365	Structured_GetFirstNotationName() 70
quantity limit, getting 85	Structured_GetGeneralNamecase() 71
reference concrete syntax	Structured_GetLcnmchar() 73
case-insensitivity of element names 62	Structured_GetLcnmstrt() 74
case-sensitivity of entity names 65	Structured_GetNextElementName() 77
reserved names	Structured_GetNextEntityName() 78
getting 87	Structured_GetNextNotationName() 80
writing to a document instance 379	Structured_GetNotationDef() 82
writing to the DTD 379	Structured_GetOptionalFeature() 83
retrieving default entity definition 58	Structured_GetQuantity() 85
Sr_Convert() 17	Structured_GetReservedName() 87
start-tag, notifying FrameMaker about writing	Structured_GetUcnmchar() 89
out a 339	Structured_GetUcnmstrt() 90
Sw_Convert() 17	Structured_IsAttrFixed() 91, 93, 97
system identifier	Structured_IsAttrNameToken() 94
getting a graphic or equation for export 316	Structured_IsAttrValSpecified() 95
setting a graphic or equation for export 366	Structured_IsIdAttr() 96
tasks requiring clients 12	StructuredAttrDeclaredValueT 384

StructuredAttrDefaultValueT 385	Sw_GetGfxEntityName() 311
StructuredAttrDefsT 411	Sw_GetGfxEntityType() 312
StructuredAttrDefT 411	Sw_GetGfxNotation() 314
StructuredAttrValsT 413	Sw_GetGfxPubId() 315
StructuredAttrValT 412	Sw_GetGfxSysId() 316
StructuredContentTokenT 385	Sw_GetObjType() 317
StructuredContentTypeT 386	Sw_GetParentConvObj() 318
StructuredDelimiterTypeT 387	Sw_GetPI() 319
StructuredElementDefT 413	Sw_GetPrivateData() 320
StructuredEntityDefT 414	Sw_GetProcessingFlags() 323
StructuredEntityScopeT 388	Sw_GetPropVal() 324
StructuredEntityTypeT 389	Sw_GetPropVals() 326
StructuredFeatureTypeT 390	Sw_GetSessionProps() 329
StructuredNotationDefT 415	Sw_GetStructuredGi() 330
StructuredQuantityTypeT 391	Sw_GetStructuredText() 332
StructuredReservedNameT 392	Sw_IsExportingToXml() 334
SUBDOC 84	Sw_IsGeneralEntityNameDefined() 335
Sw_CancelCurBatchFile() 281	Sw_IsGeneralEntityNameUsed() 336
Sw_CancelOperation() 282	Sw_NotifyEndTag() 337
Sw_Convert() 17, 283	Sw_NotifyGeneralEntityDef() 338
Sw_DeallocateSessionProps() 284	Sw_NotifyStartTag() 339
Sw_DeallocateTextItems() 285	Sw_ScanElem() 340
Sw_EventHandler() 259, 286	Sw_SetAttrVal() 345
Sw_GetAssociatedEvent() 288	Sw_SetAttrVals() 347
Sw_GetAttrVal() 290	Sw_SetBookCompEntityFilePath() 350
Sw_GetBookCompEntityFilePath() 295	Sw_SetBookCompPi() 351
Sw_GetBookCompPi() 296	Sw_SetBookPi() 352
Sw_GetBookPi() 297	Sw_SetColSpecs() 354
Sw_GetChildConvObj() 298	Sw_SetEntityName() 356
Sw_GetColSpecs() 300	Sw_SetExportFileFormat() 357
Sw_GetCurConvObj() 302	Sw_SetExportFilePath() 359
Sw_GetCurConvObjOfType() 303	Sw_SetGfxDataAttrVals() 360
Sw_GetEntityName() 306	Sw_SetGfxEntityName() 361
Sw_GetExportFileFormat() 307	Sw_SetGfxEntityType() 362
Sw_GetExportFilePath() 309	Sw_SetGfxNotation() 364
Sw_GetGfxDataAttrVals() 310	Sw_SetGfxPubId() 365

Sw_SetGfxSysId() 366
 Sw_SetPI() 367
 Sw_SetPrivateData() 368
 Sw_SetProcessingFlags() 369
 Sw_SetSessionProps() 370
 Sw_SetStructuredGi() 372
 Sw_SetStructuredText() 373
 Sw_SetTableScanOrder() 374
 Sw_WriteAttrSpec() 376
 Sw_WriteDelimiter() 377
 Sw_WriteReservedName() 379
 Sw_WriteString() 380
 SwConvObjT 424
 SwEventT 425
 SwEventTypeT 404
 switch statements in event handlers 23
 SwLocationT 406
 SwObjTypeT 407
 SwSessionPropsT 425
 SwTextlItemsT 426
 SwTextlItemT 426
 SwTextlItemTypeT 408
 system identifier
 for exporting graphics and equations
 getting 316
 setting 366

T

tables

adding rows to during import 99
 cells
 marking as used 226
 testing for forcing of new rows 172
 testing for used 103
 colspecs
 copying 239

copying a list of 240
 freeing 250
 getting by column name from a list 262
 getting by column name on import 127
 getting by column number from a list 261
 getting by column number on import 126
 getting for export 300
 getting on import 124
 setting for export 354
 setting import 195
 updating an existing colspec in a list 275
 column-major 12
 element name for inserted part
 getting 146
 setting 209
 rows
 adding on import 99
 marking as used 227
 testing for use 182
 scan order, setting for export 375
 spanspecs
 copying 245
 copying a list of 246
 getting by name on import 131
 getting on import 169
 inserting a new spanspec in a list 276
 setting on import 224
 updating an existing spanspec in a list 276
 straddles
 copying 247
 copying lists of 248
 freeing by name 258
 getting on import 170
 inserting a new straddle in a list of 277
 setting on import 225

tag

- getting FrameMaker element 140
- setting FrameMaker element 203
- tag omission 50
- tasks requiring import/export clients 12
- template document ID, getting 173
- text
 - getting FrameMaker on export 332
 - getting FrameMaker on import 143
 - setting FrameMaker on export 373
 - setting FrameMaker on import 205
- text insets
 - filepath
 - getting 174
 - setting 229
 - flow
 - getting 176
 - setting 231
 - page space
 - getting 177, 179, 232
 - setting 234

U

- UCNMCHAR parameter 89
- UCNMSTRT parameter 90
- using constants in import/export clients 22
- using global declarations in import/export clients 22

V

- variables
 - nonelement name
 - getting 180
 - setting 235

X

- XML, exporting to 334