VMS

digital

VMS Compound Document Architecture Manual

Order Number: AA-MG30A-TE

VMS Compound Document Architecture Manual

Order Number: AA–MG30A–TE

December 1988

This manual describes the DIGITAL Compound Document Architecture and the tools that support it.

Revision/Update Information: This is a new manual.

Software Version: VMS Version 5.1

digital equipment corporation
maynard, massachusetts

December 1988

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

© Digital Equipment Corporation 1988.

All Rights Reserved.
Printed in U.S.A.

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

CDA	MASSBUS	VAX RMS
DDIF	PrintServer 40	VAXstation
DEC	Q-bus	VMS
DECnet	ReGIS	VT
DECUS	ULTRIX	XUI
DECwindows	UNIBUS	
DIGITAL	VAX	
LN03	VAXcluster	digital™

PostScript is a registered trademark of Adobe Systems, Inc.

ZK4737

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by DIGITAL. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use DIGITAL-supported devices, such as the LN03 laser printer and PostScript printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

Contents

PREFACE xxiii

CHAPTER 1 INTRODUCTION 1–1

1.1 COMPOUND DOCUMENTS 1–1

1.2 OVERVIEW OF THE COMPOUND DOCUMENT ARCHITECTURE 1–2
1.2.1 The DIGITAL Document Interchange Format 1–3
1.2.2 The CDA Toolkit 1–3
1.2.3 The CDA Converter Architecture 1–3

1.3 DOCUMENT PROCESSING CONCEPTS 1–4
1.3.1 Document Structure 1–4
1.3.2 Document Layout 1–4
1.3.3 Logical Structure and Layout 1–6
1.3.4 Structured and Unstructured Markup Systems 1–6
1.3.5 Interactive and Batch Processing 1–7

1.4 SEPARATION OF LAYOUT FROM CONTENT 1–7
1.4.1 Replacement of Layout 1–7

CHAPTER 2 CDA CONVERTER ARCHITECTURE 2–1

2.1 CDA CONVERTER 2–1
2.1.1 Components of a Converter 2–2
2.1.2 DCL CONVERT/DOCUMENT Command 2–3
2.1.3 CONVERT Routine 2–4

2.2 DDIF VIEWER 2–7
2.2.1 DCL VIEW Command 2–7

2.3 INPUT FORMATS 2–9
2.3.1 DDIF Front End 2–9
2.3.1.1 Data Mapping • 2–9
2.3.1.2 Data Loss • 2–9

2.3.1.3 External File References • **2–9**
2.3.1.4 Document Syntax Errors • **2–9**
2.3.2 Text Front End **2–10**
2.3.2.1 Data Mapping • **2–10**
2.3.2.2 Data Loss • **2–10**
2.3.2.3 External File References • **2–10**
2.3.2.4 Document Syntax Errors • **2–10**

2.4 OUTPUT FORMATS **2–10**
2.4.1 DDIF Back End **2–10**
2.4.1.1 Data Mapping • **2–11**
2.4.1.2 Data Loss • **2–11**
2.4.2 Text Back End **2–11**
2.4.2.1 Data Mapping • **2–11**
2.4.2.2 Data Loss • **2–11**
2.4.2.3 Processing Options • **2–11**
2.4.3 PostScript Back End **2–11**
2.4.3.1 Data Mapping • **2–11**
2.4.3.2 Data Loss • **2–12**
2.4.3.3 Processing Options • **2–12**
2.4.3.4 Paper Size Processing Option • **2–12**
2.4.3.5 Paper Height Processing Option • **2–13**
2.4.3.6 Paper Width Processing Option • **2–13**
2.4.3.7 Top Margin Processing Option • **2–13**
2.4.3.8 Bottom Margin Processing Option • **2–13**
2.4.3.9 Left Margin Processing Option • **2–13**
2.4.3.10 Right Margin Processing Option • **2–13**
2.4.3.11 Paper Orientation Processing Option • **2–14**
2.4.3.12 Eight Bit Output Processing Option • **2–14**
2.4.3.13 Output Buffer Size Processing Option • **2–14**
2.4.3.14 Soft Directives Processing Option • **2–14**
2.4.3.15 Word Wrap Processing Option • **2–14**
2.4.3.16 Page Wrap Processing Option • **2–15**
2.4.3.17 Layout Processing Option • **2–15**
2.4.4 Analysis Back End **2–15**

CHAPTER 3 OVERVIEW OF DDIF **3–1**

3.1 DOCUMENT CONTENT **3–1**
3.1.1 Document Hierarchy **3–2**
3.1.2 Document Root **3–3**
3.1.3 Document Descriptor **3–4**
3.1.4 Document Header **3–4**

3.1.5	**Root Segment**	**3–4**
3.1.5.1	Text Content • **3–5**	
3.1.5.2	Graphics Content • **3–5**	
3.1.5.3	Image Content • **3–5**	
3.1.5.4	Computed Content • **3–5**	
3.1.5.5	Restricted Content • **3–6**	
3.1.5.6	Private Data • **3–6**	
3.1.6	**Relationships in Revisable Documents**	**3–6**
3.1.6.1	Attribute Inheritance • **3–8**	
3.1.6.2	Generic Types • **3–8**	
3.1.6.3	Generic Content • **3–8**	
3.1.6.4	References to Generic Types • **3–9**	
3.1.6.5	References to Generic Content • **3–9**	
3.1.7	**Example of Document Content**	**3–9**
3.2	**DOCUMENT LAYOUT**	**3–14**
3.2.1	**Page Description**	**3–14**
3.2.2	**Page Set**	**3–14**
3.2.3	**Page Layout**	**3–15**
3.2.4	**Galley**	**3–15**
3.2.5	**Implementation of Layout Separation**	**3–15**
3.2.5.1	Wrap Attributes • **3–16**	
3.2.5.2	Layout Attributes • **3–16**	
3.2.6	**Content Streams in Layout**	**3–16**

CHAPTER 4 OVERVIEW OF THE CDA TOOLKIT **4–1**

4.1	**CDA TOOLKIT ROUTINES TERMINOLOGY**	**4–1**
4.2	**FILE MANAGEMENT**	**4–2**
4.3	**STREAM MANAGEMENT**	**4–3**
4.4	**ROOT AGGREGATE MANAGEMENT**	**4–3**
4.5	**AGGREGATE MANAGEMENT**	**4–4**

4.6	ITEM ACCESS	4–5
4.7	DOCUMENT CONVERSION	4–9
4.7.1	Document Transfer	4–10
4.7.2	Aggregate Transfer	4–10
4.8	CDA CONVERTERS	4–13

CHAPTER 5 WRITING CONVERTER FRONT AND BACK ENDS 5–1

5.1	DOCUMENT CONVERSION	5–1
5.2	FRONT END	5–2
5.2.1	DDIF$READ_*format* Entry Point	5–4
5.2.2	*Get-Aggregate* Entry Point	5–6
5.2.3	*Get-Position* Entry Point	5–7
5.2.4	*Close* Entry Point	5–8
5.2.5	Front End Document-Method Conversion	5–8
5.2.5.1	DDIF$READ_*format* Routine • **5–9**	
5.2.5.2	*Get-Aggregate* Routine • **5–10**	
5.2.5.3	*Get-Position* Routine • **5–10**	
5.2.5.4	*Close* Routine • **5–10**	
5.2.6	Front End Aggregate-Method Conversion	5–10
5.2.6.1	*Get-Aggregate* Routine • **5–12**	
5.2.6.2	*Get-Position* Routine • **5–12**	
5.2.6.3	*Close* Routine • **5–13**	
5.3	USER-SUPPLIED INPUT PROCEDURES	5–13
5.4	BACK END ROUTINE	5–14
5.4.1	DDIF$WRITE_*format* Entry Point	5–14
5.4.2	User-Supplied Output Procedures	5–17

CHAPTER 6 DDIF STRUCTURES 6–1

6.1 DDIF DOCUMENT STRUCTURE OVERVIEW 6–1

6.2 GENERIC AGGREGATE ITEMS 6–2

6.3 DOCUMENT ROOT AGGREGATE 6–2

6.4 DOCUMENT DESCRIPTOR 6–3

6.5 DOCUMENT HEADER 6–4

6.6 DOCUMENT CONTENT 6–6
6.6.1 Content Categories 6–8
6.6.2 Segment Tags 6–9
6.6.3 Presentation Attributes of Content 6–9

6.7 TEXT CONTENT 6–9
6.7.1 Latin1 Text Content 6–10
6.7.2 General Text Content 6–10

6.8 DIRECTIVES 6–11
6.8.1 Hard Directive 6–11
6.8.2 Soft Directive 6–11
6.8.3 Directive Values 6–12
6.8.4 Hard Value Directive 6–13
6.8.5 Soft Value Directive 6–14

6.9 BEZIER CURVE CONTENT 6–15

6.10 POLYLINE CONTENT 6–16

6.11 ARC CONTENT 6–18

6.12 FILL AREA SET CONTENT 6–20

6.13	**IMAGE CONTENT**	**6–21**
6.14	**CONTENT REFERENCE AGGREGATE**	**6–22**
6.15	**RESTRICTED CONTENT**	**6–22**
6.15.1	**External (PDL) Content**	**6–22**
6.15.2	**Private Content**	**6–24**
6.16	**LAYOUT GALLEY**	**6–25**
6.17	**EXTERNAL REFERENCE**	**6–28**
6.18	**IMAGE DATA UNIT**	**6–30**
6.19	**COMPOSITE PATH**	**6–32**
6.20	**SEGMENT ATTRIBUTES**	**6–35**
6.20.1	**General Segment Attributes**	**6–35**
6.20.2	**Computed Content Attributes**	**6–37**
6.20.2.1	Copied and Remote Computed Content • **6–38**	
6.20.2.2	Variable Computed Content • **6–38**	
6.20.2.3	Cross-Reference Computed Content • **6–38**	
6.20.2.4	Function Computed Content • **6–38**	
6.20.3	**Structure Attributes**	**6–39**
6.20.4	**Language**	**6–39**
6.20.5	**Legend**	**6–40**
6.20.6	**Measurement**	**6–40**
6.20.7	**Alternate Presentation**	**6–41**
6.20.8	**Layout**	**6–41**
6.20.8.1	Galley-Based Layout • **6–42**	
6.20.8.2	Path-Based Layout • **6–42**	
6.20.8.3	Position-Relative Layout • **6–45**	
6.20.8.4	Text Position • **6–46**	
6.20.9	**Font Definitions**	**6–47**
6.20.10	**Pattern Definitions**	**6–47**
6.20.11	**Path Definitions**	**6–47**
6.20.12	**Line-Style Definitions**	**6–47**
6.20.13	**Content Definitions**	**6–47**
6.20.14	**Type Definitions**	**6–48**
6.20.15	**Text Attributes**	**6–48**
6.20.15.1	Text Mask Pattern • **6–48**	

6.20.15.2 Text Font • **6–48**
6.20.15.3 Text Rendition • **6–49**
6.20.15.4 Text Size • **6–50**
6.20.15.5 Text Direction • **6–51**
6.20.15.6 Text Character Decimal Alignment • **6–51**
6.20.15.7 Text Leader Attributes • **6–51**
6.20.15.8 Text Kerning • **6–52**
6.20.16 Line Attributes — **6–52**
6.20.17 Marker Attributes — **6–55**
6.20.18 Galley Attributes — **6–55**
6.20.19 Image Attributes — **6–56**
6.20.20 Image Component Space Attributes — **6–58**
6.20.21 Frame Parameters — **6–60**
6.20.21.1 Frame Flags • **6–60**
6.20.21.2 Frame Bounding Box • **6–61**
6.20.21.3 Frame Outline • **6–61**
6.20.21.4 Frame Clipping • **6–62**
6.20.21.5 Frame Position • **6–62**
6.20.21.5.1 Fixed Frame Parameters • 6–63
6.20.21.5.2 Inline Frame Parameters • 6–63
6.20.21.5.3 Galley Frame Parameters • 6–63
6.20.21.5.4 Margin Frame Parameters • 6–64
6.20.21.6 Frame Content Transformation • **6–65**
6.20.22 Item Change List — **6–65**
6.20.23 Segment Attribute Items and Types — **6–66**

6.21 CONTENT DEFINITION 6–69

6.22 FONT DEFINITION 6–70

6.23 LINE-STYLE DEFINITION 6–71

6.24 PATH DEFINITION 6–72

6.25 PATTERN DEFINITION 6–73

6.26 SEGMENT BINDING 6–75
6.26.1 Counter Variable Values — **6–76**
6.26.2 Computed Variable Values — **6–77**
6.26.3 List Variable Values — **6–77**
6.26.4 Segment Binding Items and Types — **6–78**

6.27	TYPE DEFINITION	6–78
6.28	COUNTER STYLE	6–79
6.29	OCCURRENCE DEFINITION	6–80
6.30	RECORD DEFINITION	6–81
6.31	IMAGE LOOKUP TABLE ENTRY	6–82
6.32	TRANSFORMATION	6–83
6.33	GENERIC LAYOUT	6–84
6.34	SPECIFIC LAYOUT	6–85
6.35	WRAP ATTRIBUTES	6–86
6.36	LAYOUT ATTRIBUTES	6–88
6.37	GALLEY ATTRIBUTES	6–91
6.38	PAGE DESCRIPTION	6–93
6.39	PAGE LAYOUT	6–94
6.40	PAGE SELECT	6–96
6.41	TAB STOP	6–97

CDA REFERENCE SECTION

CDA$AGGREGATE_TYPE_TO_OBJECT_ID	CDA–3
CDA$CLOSE_FILE	CDA–5
CDA$CLOSE_STREAM	CDA–7
CDA$CLOSE_TEXT_FILE	CDA–8
CDA$CONVERT	CDA–9
CDA$CONVERT_AGGREGATE	CDA–26
CDA$CONVERT_DOCUMENT	CDA–29
CDA$CONVERT_POSITION	CDA–31
CDA$COPY_AGGREGATE	CDA–33
CDA$CREATE_AGGREGATE	CDA–35
CDA$CREATE_FILE	CDA–37
CDA$CREATE_ROOT_AGGREGATE	CDA–42
CDA$CREATE_STREAM	CDA–46
CDA$CREATE_TEXT_FILE	CDA–51
CDA$DELETE_AGGREGATE	CDA–54
CDA$DELETE_ROOT_AGGREGATE	CDA–56
CDA$ENTER_SCOPE	CDA–57
CDA$ERASE_ITEM	CDA–68
CDA$FIND_DEFINITION	CDA–70
CDA$FIND_TRANSFORMATION	CDA–73
CDA$FLUSH_STREAM	CDA–75
CDA$GET_AGGREGATE	CDA–77
CDA$GET_ARRAY_SIZE	CDA–80
CDA$GET_DOCUMENT	CDA–82
CDA$GET_EXTERNAL_ENCODING	CDA–84
CDA$GET_STREAM_POSITION	CDA–86
CDA$GET_TEXT_POSITION	CDA–89
CDA$INSERT_AGGREGATE	CDA–91
CDA$LEAVE_SCOPE	CDA–94
CDA$LOCATE_ITEM	CDA–96
CDA$NEXT_AGGREGATE	CDA–99
CDA$OBJECT_ID_TO_AGGREGATE_TYPE	CDA–101
CDA$OPEN_CONVERTER	CDA–103
CDA$OPEN_FILE	CDA–106
CDA$OPEN_STREAM	CDA–112
CDA$OPEN_TEXT_FILE	CDA–116
CDA$PRUNE_AGGREGATE	CDA–119
CDA$PRUNE_POSITION	CDA–121
CDA$PUT_AGGREGATE	CDA–123
CDA$PUT_DOCUMENT	CDA–126

CDA$READ_TEXT_FILE		**CDA–128**
CDA$REMOVE_AGGREGATE		**CDA–130**
CDA$STORE_ITEM		**CDA–131**
CDA$WRITE_TEXT_FILE		**CDA–137**

APPENDIX A VMS SUPPORT FOR CDA IN DECWINDOWS **A–1**

A.1	**VMS COMMANDS AND UTILITIES**	**A–1**
A.1.1	**Displaying RMS File Tags**	**A–2**
A.1.1.1	DIRECTORY/FULL • **A–2**	
A.1.1.2	ANALYZE/RMS_FILE • **A–2**	
A.1.2	**Creating RMS File Tags**	**A–3**
A.1.3	**Preserving RMS File Tags and DDIF Semantics**	**A–4**
A.1.3.1	COPY Command • **A–4**	
A.1.3.2	VMS Mail Utility • **A–5**	
A.1.4	**APPEND Command**	**A–5**
A.2	**DDIF SUPPORT IN A HETEROGENEOUS ENVIRONMENT**	**A–6**
A.2.1	**EXCHANGE/NETWORK Command**	**A–6**
A.2.2	**Using the COPY Command in a Heterogeneous Environment**	**A–6**
A.2.3	**VMS Mail Utility in a Heterogeneous Environment**	**A–6**
A.3	**VMS RMS INTERFACE CHANGES**	**A–7**
A.3.1	**Programming Interface for File Tagging**	**A–7**
A.3.2	**Accessing a Tagged File**	**A–10**
A.3.2.1	File Accesses That Do Not Sense Tags • **A–11**	
A.3.2.2	File Accesses That Sense Tags • **A–11**	
A.3.3	**Preserving Tags**	**A–13**
A.4	**DISTRIBUTED FILE SYSTEM SUPPORT FOR DDIF TAGGED FILES**	**A–14**
A.5	**VMS RMS ERRORS**	**A–14**

APPENDIX B CDA TOOLKIT EXAMPLE PROGRAM **B–1**

APPENDIX C TEXT FRONT END SOURCE FILE C–1

APPENDIX D DDIF AGGREGATE STRUCTURES D–1

APPENDIX E DDIF SYNTAX DIAGRAMS E–1

E.1 DDIS BUILT-IN DATA TYPES E–1

E.2 BUILT-IN OPERATORS E–3

E.3 DDIS DEFINED TYPES E–4

E.4 DDIF SYNTAX DIAGRAMS E–4

APPENDIX F DDIF FILL PATTERNS F–1

GLOSSARY OF TERMS Glossary–1

INDEX

EXAMPLES

3–1 DDIF Document Sample 3–3
3–2 DDIF Document Attribute Inheritance 3–10
A–1 Tagging a File A–9
A–2 Accessing a Tagged File A–12
B–1 Sample CDA Toolkit Program B–1
B–2 Analysis Output of DDIF File B–33

FIGURES

2–1	Stages of Document Conversion	2–1
2–2	Converter Components Diagram	2–2
3–1	Document Hierarchy	3–2
3–2	Typical DDIF Document	3–4
3–3	Illustration of Inheritance Example Document	3–13
4–1	Document Segment Aggregate	4–6
5–1	Document Conversion Flowchart	5–3
6–1	Compound Document Structure	6–1
E–1	DDIF Document Syntax Diagram	E–5
E–2	Document Descriptor Syntax Diagram	E–5
E–3	Document Header Syntax Diagram	E–5
E–4	Document Root Segment	E–6
E–5	Segment Primitive Syntax Diagram	E–6
E–6	Begin-Segment Syntax Diagram	E–6
E–7	Text Primitive Syntax Diagram	E–7
E–8	Text Attributes Syntax Diagram	E–7
E–9	Rendition Code Syntax Diagram	E–7
E–10	Leader Style Syntax Diagram	E–8
E–11	Text Layout Syntax Diagram	E–8
E–12	Text String Layout Syntax Diagram	E–9
E–13	Formatting Primitive Syntax Diagram	E–9
E–14	Value Directive Syntax Diagram	E–10
E–15	Directive Syntax Diagram	E–10
E–16	Escapement Directive Syntax Diagram	E–10
E–17	Variable Reset Syntax Diagram	E–11
E–18	Graphics Primitive Syntax Diagram	E–11
E–19	Polyline Syntax Diagram	E–11
E–20	Cubic Bézier Syntax Diagram	E–12
E–21	Arc Syntax Diagram	E–12
E–22	Fill Area Set Syntax Diagram	E–12
E–23	Line Attributes Syntax Diagram	E–13
E–24	Line Style Number Syntax Diagram	E–13
E–25	Line End Number Syntax Diagram	E–13
E–26	Line Join Syntax Diagram	E–14
E–27	Marker Attributes Syntax Diagram	E–14
E–28	Marker Number Syntax Diagram	E–14

E–29	Image Primitive Syntax Diagram	E–15
E–30	Image Coding Attributes Syntax Diagram	E–15
E–31	Image Attributes Syntax Diagram	E–16
E–32	Image Lookup Table Data Syntax Diagram	E–17
E–33	Image Component Space Attributes Syntax Diagram	E–17
E–34	Restricted Content Syntax Diagram	E–17
E–35	Content Reference Primitive Syntax Diagram	E–18
E–36	Content Reference Syntax Diagram	E–18
E–37	Bounding Box Syntax Diagram	E–18
E–38	Color Syntax Diagram	E–18
E–39	Red/Green/Blue Syntax Diagram	E–19
E–40	Compute Definition Syntax Diagram	E–19
E–41	Cross Reference Syntax Diagram	E–19
E–42	Escapement Syntax Diagram	E–20
E–43	External Reference Syntax Diagram	E–20
E–44	Font Definition Syntax Diagram	E–20
E–45	Format Syntax Diagram	E–21
E–46	Frame Parameters Syntax Diagram	E–21
E–47	Inline Frame Parameters Syntax Diagram	E–21
E–48	Galley Frame Parameters Syntax Diagram	E–22
E–49	Galley Vertical Position Syntax Diagram	E–22
E–50	Margin Frame Parameters Syntax Diagram	E–22
E–51	Margin Horizontal Position Syntax Diagram	E–23
E–52	Function Link Syntax Diagram	E–23
E–53	External Reference Index Syntax Diagram	E–23
E–54	Language Index Syntax Diagram	E–23
E–55	Content Definition Syntax Diagram	E–24
E–56	Label Syntax Diagram	E–24
E–57	Label Types Syntax Diagram	E–24
E–58	ASCII String Syntax Diagram	E–24
E–59	Variable Label Syntax Diagram	E–25
E–60	Legend Units Syntax Diagram	E–25
E–61	Angle Syntax Diagram	E–25
E–62	AngleRef Syntax Diagram	E–25
E–63	Measurement Syntax Diagram	E–25
E–64	Position Syntax Diagram	E–26
E–65	Ratio Syntax Diagram	E–26
E–66	Right Angle Syntax Diagram	E–26
E–67	Size Syntax Diagram	E–26
E–68	X-Coordinate Syntax Diagram	E–27

E–69	Y-Coordinate Syntax Diagram	E–27
E–70	Measurement Units Syntax Diagram	E–27
E–71	Named Value Syntax Diagram	E–27
E–72	Value Data Syntax Diagram	E–28
E–73	Named Value List Syntax Diagram	E–28
E–74	Font Number Syntax Diagram	E–28
E–75	Marker Number Syntax Diagram	E–28
E–76	Path Number Syntax Diagram	E–29
E–77	Pattern Number Syntax Diagram	E–29
E–78	Path Definition Syntax Diagram	E–29
E–79	Composite Path Syntax Diagram	E–29
E–80	Arc Path Syntax Diagram	E–30
E–81	Cubic Bézier Path Syntax Diagram	E–30
E–82	Line Definition Syntax Diagram	E–30
E–83	Polyline Path Syntax Diagram	E–31
E–84	Pattern Definition Syntax Diagram	E–31
E–85	Standard Pattern Syntax Diagram	E–31
E–86	Reference Syntax Diagram	E–31
E–87	Segment Attributes Syntax Diagram	E–32
E–88	Segment Type Definition Syntax Diagram	E–32
E–89	Structure Definition Syntax Diagram	E–33
E–90	Occurrence Definition Syntax Diagram	E–33
E–91	Structure Element Syntax Diagram	E–33
E–92	Tag Syntax Diagram	E–33
E–93	Category Tag Syntax Diagram	E–34
E–94	Conformance Tag Syntax Diagram	E–34
E–95	Named Value Tag Syntax Diagram	E–34
E–96	Segment Tag Syntax Diagram	E–34
E–97	Storage System Tag Syntax Diagram	E–34
E–98	Stream Tag Syntax Diagram	E–35
E–99	Transformation Syntax Diagram	E–35
E–100	Variable Binding Syntax Diagram	E–35
E–101	Counter Definition Syntax Diagram	E–36
E–102	Layout Object Type Syntax Diagram	E–36
E–103	Expression Syntax Diagram	E–36
E–104	Counter Style Syntax Diagram	E–37
E–105	String Expression Syntax Diagram	E–37
E–106	Record List Syntax Diagram	E–37
E–107	Record Definition Syntax Diagram	E–38
E–108	Generic Layout Syntax Diagram	E–38

E–109	Page Description Syntax Diagram	E–38
E–110	Page Set Syntax Diagram	E–39
E–111	Page Layout Syntax Diagram	E–39
E–112	Layout Primitive Syntax Diagram	E–39
E–113	Layout Galley Syntax Diagram	E–40
E–114	Galley Attributes Syntax Diagram	E–40
E–115	Specific Layout Syntax Diagram	E–40
E–116	Wrap Attributes Syntax Diagram	E–41
E–117	Layout Attributes Syntax Diagram	E–41
E–118	Break Criteria Syntax Diagram	E–42
E–119	General Measure Syntax Diagram	E–42
E–120	General Size Syntax Diagram	E–42
E–121	Tab Stop List Syntax Diagram	E–42
E–122	Tab Stop Syntax Diagram	E–43
F–1	CDA Fill Patterns	F–6

TABLES

1–1	Layout Terminology	1–5
2–1	Converter Format Keywords	2–4
3–1	Relationships in Revisable Documents	3–6
4–1	Routines Terminology	4–1
4–2	Item Data Types	4–6
5–1	Top-Level Aggregate Types	5–12
6–1	Generic Aggregate Items	6–2
6–2	Document Root Aggregate (DDIF$_DDF)	6–2
6–3	Document Descriptor Aggregate (DDIF$_DSC)	6–4
6–4	Document Header Aggregate (DDIF$_DHD)	6–6
6–5	Document Segment Aggregate (DDIF$_SEG)	6–8
6–6	Latin1 Text Content Aggregate (DDIF$_TXT)	6–10
6–7	General Text Content Aggregate (DDIF$_GTX)	6–10
6–8	Character Set Identifiers	6–10
6–9	Hard Directive Aggregate (DDIF$_HRD)	6–11
6–10	Soft Directive Aggregate (DDIF$_SFT)	6–11
6–11	Directive Values	6–12
6–12	Hard Value Directive Aggregate (DDIF$_HRV)	6–14
6–13	Soft Value Directive Aggregate (DDIF$_SFV)	6–15
6–14	Bézier Curve Aggregate (DDIF$_BEZ)	6–16
6–15	Polyline Aggregate (DDIF$_LIN)	6–18
6–16	Arc Content Aggregate (DDIF$_ARC)	6–19

6–17	Fill Area Set Content Aggregate (DDIF$_FAS)	6–21
6–18	Image Content Aggregate (DDIF$_IMG)	6–22
6–19	Content Reference Aggregate (DDIF$_CRF)	6–22
6–20	External Content Aggregate (DDIF$_EXT)	6–23
6–21	Private Content Aggregate (DDIF$_PVT)	6–25
6–22	Layout Galley Aggregate (DDIF$_GLY)	6–27
6–23	Object Identifier Table	6–28
6–24	External Reference Aggregate (DDIF$_ERF)	6–29
6–25	Image Data Unit Aggregate (DDIF$_IDU)	6–31
6–26	Composite Path Aggregate (DDIF$_PTH)	6–34
6–27	Normal Alignment	6–45
6–28	Line Style	6–52
6–29	Segment Attributes Aggregate (DDIF$_SGA)	6–66
6–30	Content Definition Aggregate (DDIF$_CTD)	6–70
6–31	Font Definition Aggregate (DDIF$_FTD)	6–71
6–32	Line-Style Definition Aggregate (DDIF$_LSD)	6–72
6–33	Path Definition Aggregate (DDIF$_PHD)	6–73
6–34	Pattern Definition Aggregate (DDIF$_PTD)	6–75
6–35	Segment Binding Aggregate (DDIF$_SGB)	6–78
6–36	Type Definition Aggregate (DDIF$_TYD)	6–79
6–37	Counter Style Aggregate (DDIF$_CTS)	6–80
6–38	Occurrence Definition Aggregate (DDIF$_OCC)	6–81
6–39	Record Definition Aggregate (DDIF$_RCD)	6–82
6–40	RGB Lookup Table Entry Aggregate (DDIF$_RGB)	6–82
6–41	Transformation Aggregate (DDIF$_TRN)	6–84
6–42	Generic Layout 1 Aggregate (DDIF$_LG1)	6–85
6–43	Specific Layout 1 Aggregate (DDIF$_LS1)	6–86
6–44	Wrap Attributes 1 Aggregate (DDIF$_LW1)	6–88
6–45	Layout Attributes 1 Aggregate (DDIF$_LL1)	6–91
6–46	Galley Attributes Aggregate (DDIF$_GLA)	6–92
6–47	Page Description Aggregate (DDIF$_PGD)	6–94
6–48	Page Layout Aggregate (DDIF$_PGL)	6–96
6–49	Page Select Aggregate (DDIF$_PGS)	6–97
6–50	Tab Stop Aggregate (DDIF$_TBS)	6–98
A–1	Tag Support Item Codes	A–7
D–1	Document Root Aggregate (DDIF$_DDF)	D–1
D–2	Document Descriptor Aggregate (DDIF$_DSC)	D–1
D–3	Document Header Aggregate (DDIF$_DHD)	D–1
D–4	Document Segment Aggregate (DDIF$_SEG)	D–2
D–5	Latin1 Text Content Aggregate (DDIF$_TXT)	D–2

D–6	General Text Content Aggregate (DDIF$_GTX)	D–2
D–7	Hard Directive Aggregate (DDIF$_HRD)	D–3
D–8	Soft Directive Aggregate (DDIF$_SFT)	D–3
D–9	Hard Value Directive Aggregate (DDIF$_HRV)	D–3
D–10	Soft Value Directive Aggregate (DDIF$_SFV)	D–3
D–11	Bézier Curve Aggregate (DDIF$_BEZ)	D–4
D–12	Polyline Aggregate (DDIF$_LIN)	D–4
D–13	Arc Content Aggregate (DDIF$_ARC)	D–4
D–14	Fill Area Set Content Aggregate (DDIF$_FAS)	D–5
D–15	Image Content Aggregate (DDIF$_IMG)	D–5
D–16	Content Reference Aggregate (DDIF$_CRF)	D–5
D–17	External Content Aggregate (DDIF$_EXT)	D–5
D–18	Private Content Aggregate (DDIF$_PVT)	D–6
D–19	Layout Galley Aggregate (DDIF$_GLY)	D–6
D–20	External Reference Aggregate (DDIF$_ERF)	D–7
D–21	Image Data Unit Aggregate (DDIF$_IDU)	D–7
D–22	Composite Path Aggregate (DDIF$_PTH)	D–7
D–23	Segment Attributes Aggregate (DDIF$_SGA)	D–8
D–24	Content Definition Aggregate (DDIF$_CTD)	D–12
D–25	Font Definition Aggregate (DDIF$_FTD)	D–12
D–26	Line Style Definition Aggregate (DDIF$_LSD)	D–12
D–27	Path Definition Aggregate (DDIF$_PHD)	D–13
D–28	Pattern Definition Aggregate (DDIF$_PTD)	D–13
D–29	Segment Binding Aggregate (DDIF$_SGB)	D–13
D–30	Type Definition Aggregate (DDIF$_TYD)	D–14
D–31	Counter Style Aggregate (DDIF$_CTS)	D–14
D–32	Occurrence Definition Aggregate (DDIF$_OCC)	D–14
D–33	Record Definition Aggregate (DDIF$_RCD)	D–14
D–34	RGB Lookup Table Entry Aggregate (DDIF$_RGB)	D–15
D–35	Transformation Aggregate (DDIF$_TRN)	D–15
D–36	Generic Layout 1 Aggregate (DDIF$_LG1)	D–15
D–37	Specific Layout 1 Aggregate (DDIF$_LS1)	D–15
D–38	Wrap Attributes 1 Aggregate (DDIF$_LW1)	D–16
D–39	Layout Attributes 1 Aggregate (DDIF$_LL1)	D–16
D–40	Galley Attributes Aggregate (DDIF$_GLA)	D–17
D–41	Page Description Aggregate (DDIF$_PGD)	D–17
D–42	Page Layout Aggregate (DDIF$_PGL)	D–17
D–43	Page Select Aggregate (DDIF$_PGS)	D–18
D–44	Tab Stop Aggregate (DDIF$_TBS)	D–18
E–1	DDIS Built-In Primitives	E–1

E–2	DDIS Built-In Constructors	E–3
E–3	DDIS Built-In Operators	E–3
E–4	DDIS Defined Types	E–4
F–1	DDIF Fill Patterns	F–1

Preface

This manual is designed to introduce the concepts and tools associated with the DIGITAL Compound Document Architecture (CDA), including the DIGITAL Document Interchange Format (DDIF). Using representation formats such as this, the Compound Document Architecture provides a method for manipulating files that contain a number of integrated components.

The tools associated with the Compound Document Architecture include the CDA Toolkit, the CDA Converter, and the DDIF Viewer. The CDA Toolkit is a collection of routines that support the creation of CDA applications. The CDA Converter is used to convert files of a specified input format to a specified output format. The DDIF Viewer is used to display DDIF-encoded files on a workstation display or character cell terminal.

All of the following products support CDA-encoded files. If you only intend to manipulate DDIF files, and do not have an interest in the particulars of the file format, you can use any one of these products to manipulate a CDA-encoded file.

DECpaint	PrintScreen	CardFiler
GKS	PHIGS	DDIF Viewer
DECwindows MAIL	Image Services Library	Converters

Intended Audience

This manual is intended for system and application programmers who want to make use of DIGITAL's new Compound Document Architecture. Some knowledge of the tasks and terminology associated with document typesetting is helpful.

Document Structure

This manual consists of two parts: an introductory section and a reference section. The first part of this manual provides general user and application programmer information regarding the Compound Document Architecture (CDA), the DIGITAL Document Interchange Format (DDIF), the CDA Converter, and the CDA Toolkit routines. The CDA reference section describes each of the CDA Toolkit routines individually.

The chapters are summarized as follows:

- Chapter 1 provides an overview of the components of the Compound Document Architecture.
- Chapter 2 discusses the use of the CDA Converter, the DDIF Viewer, and the various supported file-encoding formats.
- Chapter 3 provides an overview of the concepts incorporated in the DIGITAL Document Interchange Format (DDIF) architecture.

- Chapter 4 discusses the CDA Toolkit routines.
- Chapter 5 provides suggestions and guidelines that should be followed when creating CDA Converter front and back ends.
- Chapter 6 describes the structure and encoding of each aggregate supported by the DDIF architecture and the CDA Toolkit.

Each of the routines contained in the CDA Toolkit is described individually in the CDA reference section. The routines are documented in alphabetical order. Each routine description specifies the calling format, the encoding of the parameters, a detailed description of the function of the routine, and what condition values the routine can return.

In addition, a glossary and several appendixes are provided. The glossary defines the terminology associated with the Compound Document Architecture and the DDIF architecture. The appendixes are as follows:

- Appendix A discusses the support provided by VMS for the CDA Toolkit and the tagging of DDIF-encoded files.
- Appendix B contains an example program that uses the CDA Toolkit to create a DDIF file, and an illustration of the file created by the example program.
- Appendix C contains the source code for the Text front end to be used as an example for those wanting to develop their own front or back ends.
- Appendix D contains tables describing the items contained in each DDIF aggregate and their item encodings.
- Appendix E contains a brief overview of the DIGITAL Data Interchange Syntax (DDIS) followed by the syntax diagrams for the various constructs supported by the DDIF architecture.
- Appendix F illustrates the CDA-defined fill patterns.

Associated Documents

The Compound Document Architecture is supported by a variety of DIGITAL products. Descriptions of the support provided by each product are contained in that product's documentation. For example, GKS support for CDA is described in the GKS documentation set, and so on.

Conventions

The following conventions are used in this manual:

mouse	The term *mouse* is used to refer to any pointing device, such as a mouse, a puck, or a stylus.
MB1, MB2, MB3	MB1 indicates the left mouse button, MB2 indicates the middle mouse button, and MB3 indicates the right mouse button. (The buttons can be redefined by the user.)

PB1, PB2, PB3, PB4	PB1, PB2, PB3, and PB4 indicate buttons on the puck.
SB1, SB2	SB1 and SB2 indicate buttons on the stylus.
Ctrl/x	A sequence such as Ctrl/x indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 x	A sequence such as PF1 x indicates that you must first press and release the key labeled PF1, then press and release another key or a pointing device button.
Return	A key name is shown enclosed to indicate that you press a key on the keyboard.
. . .	In examples, a horizontal ellipsis indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
()	In format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.
[]	In format descriptions, brackets indicate that whatever is enclosed is optional; you can select none, one, or all of the choices.
{}	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.
red ink	Red ink indicates information that you must enter from the keyboard or a screen object that you must choose or click on.
boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason.
italic text	Italic text represents information that can vary in system messages (for example, Internal error *number*).
italic text	Italic text represents user-written routines (for example, *get-aggregate*).
UPPERCASE TEXT	Uppercase letters indicate that you must enter a command (for example, enter OPEN/READ).

UPPERCASE TEXT	Uppercase letters indicate the name of a CDA Toolkit routine, the name of a file, the name of a file protection code, or the abbreviation for a system privilege. Hyphens in coding examples indicate that additional arguments to the request are provided on the line that follows.
numbers	Unless otherwise noted, all numbers in the text are assumed to be decimal. Nondecimal radices—binary, octal, or hexadecimal—are explicitly indicated in the coding examples.

1 Introduction

Compound documents contain integrated components such as proportionally spaced text, synthetic graphics, and scanned or natural images. DIGITAL's Compound Document Architecture (CDA) is an open architecture that establishes a framework in which compound documents can be handled in the same easy and universal way as simple ASCII text. With CDA, you can write applications that handle compound documents easily, regardless of the environment in which you or application users are working. You do not need to be concerned with how a compound document is created and processed or how users will access the document.

The use of CDA provides numerous benefits. For example:

- CDA provides application independence. This means that applications other than the creator software can access revisable-form data and can use devices and operating environments other than the creator hardware.
- CDA makes application development easier by making the most of development resources. You can use standard CDA facilities for multiple functions (including file display and copying), thereby reducing the amount of code that has to be written.
- The use of CDA means that users can exchange documents with anyone anywhere on a DIGITAL network.

In addition, CDA satisfies the demand for inter-application data exchange by providing file conversion capabilities, including the presentation of compound document data to ASCII-oriented utilities like language compilers (using CDA filters). When an application supports CDA, it participates in the entire DIGITAL document processing environment, including live links, electronic mail of revisable compound documents, and hardware- and system-independent display and printing.

1.1 Compound Documents

The purpose of CDA is to simplify the manipulation of revisable compound documents so that complex files can be created, stored, and interchanged among users. To understand this goal, it is first important to understand the definition of a revisable compound document.

A document can be defined as a collection of data that is intended for display. A **revisable document** contains the content of a document, as well as parameters and directives that are used when creating the final form of the document. These parameters and directives specify abstract relationships between the components of the document and are used to determine the final appearance of the document (for example, line breaks and page breaks).

A revisable document does not contain page numbers, section numbers, or even a table of contents. Instead, it specifies parameters that control the creation of these elements in the final form of the document. A revisable document also does not specify the exact layout of the content of the document. Instead, it contains the basic template for the page layout and the parameters that control the way the content is arranged in the final form.

In the **final form** of a document, all the formatting decisions (such as hyphenation, line breaks, page breaks, and so on) have been resolved. Any text elements that are based on calculations, such as page numbers and section numbers, have been inserted. Also, any externally referenced document content has been included. A final form document generally does not make any distinction between document content that a user entered and document content that was generated by a formatter.

A **compound document** is a unified collection of data that can be edited, formatted, or otherwise processed as a document. Compound documents can contain a number of integrated components, including proportionally spaced text, synthetic graphics, and scanned images. That is, a compound document is a document that has the ability to contain not only text but also other integrated components. Compound documents can also contain data elements from applications such as spreadsheets.

For example, an ASCII text file is a document that comprises only text. It cannot contain integrated graphics, unless those graphics are in the form of "line art," which is represented and stored as standard text characters. A compound document, on the other hand, can include graphics that were generated by a graphics editor or scanned images. A document containing only text is considered a compound document if the document storage format has the ability to store integrated components.

A compound document also integrates the structure of a document. For example, the relationships in a chapter, that a paragraph is part of a section, and a section is part of a chapter, are integrated into a compound document that represents a chapter of a manual. This concept of structure is especially important when you are defining styles or attributes for a manual.

1.2 Overview of the Compound Document Architecture

The Compound Document Architecture provides a set of tools and utilities that simplify the treatment of compound document information. These tools and utilities are as follows:

- The DIGITAL Document Interchange Format (DDIF) for the creation, storage, and interchange of document data
- The CDA Toolkit, which is a library of callable routines that enable you to easily read, write, create, and modify compound documents
- The CDA Converter Architecture, which provides a standard CDA Converter Kernel that works with front and back ends to convert an input file of any supported format to an output file of any supported format

- Viewers, which are callable services that display formatted output data on a workstation window or character cell terminal
- Mail Utility support for sending, receiving, and displaying compound documents
- Record Management Services (RMS) support for filtering the ASCII text from a compound document for compilation, display, and printing

1.2.1 The DIGITAL Document Interchange Format

The DIGITAL Document Interchange Format (DDIF) is the format of choice for all new compound document application programs. While maintaining a strong similarity to the Office Document Architecture (ODA) and other standards, DDIF also extends the capabilities of these existing standards to reflect the growing needs of document processing.

DDIF represents structured documents that contain revisable text, graphics, and images. It supports advanced document processing features, including generic structure, independent or attached style information, logical and presentation attributes, attribute inheritance, cross-references, and "live links" (dynamic external references). DDIF is discussed in more detail in Chapter 3.

1.2.2 The CDA Toolkit

The CDA Toolkit is a collection of routines that enable you to do the following:

- Create your own CDA-conforming application
- Invoke the CDA converter from an application
- Create your own front end (to convert a document of a particular input format to its CDA in-memory representation)
- Create your own back end (to convert the CDA in-memory representation of a document to a particular output format)

The CDA Toolkit routines support a standard VMS interface and follow the VMS guidelines for condition handling. For an overview of the CDA Toolkit routines, see Chapter 4.

1.2.3 The CDA Converter Architecture

The CDA Converter Architecture defines a methodology to simplify the conversion of compound documents using a common converter kernel and a series of front and back ends. This methodology is implemented as follows:

- The conversion process is invoked through the DCL CONVERT/DOCUMENT command or through a call to the CDA Toolkit CONVERT routine.

- The converter kernel performs all the "generic" conversion functions that must be performed for every document conversion. The kernel is also responsible for invoking the appropriate front and back ends for the specified input and output file formats.
- The front end reads the input file or stream (encoded in any supported format) and converts it to its CDA in-memory representation. A front end is responsible for translating a document of a particular input format to the CDA in-memory representation. There must be a front end for every supported input format.
- The back end converts the CDA in-memory representation of the document to a particular output format and writes the data to a file or stream. A back end is responsible for translating the CDA in-memory representation of a document to a particular output format. There must be a back end for every supported output format.

The CDA Converter Architecture is discussed in more detail in Chapter 2.

1.3 Document Processing Concepts

The Compound Document Architecture is designed to simplify the processing of compound documents. The following sections discuss some of the concepts associated with compound documents and document processing.

1.3.1 Document Structure

A revisable document is an ordered hierarchy of logical elements. For example, a chapter contains sections, sections contain paragraphs and lists, and paragraphs in turn contain the text of the document. The hierarchy of these individual logical elements in a document makes up the document's **specific logical structure**.

To make it easier to share and interchange documents, it is useful to develop a set of structuring rules for documents. This set of structuring rules specifies the organization and appearance of all documents following those rules, thereby creating a **generic logical structure**. That is, a generic logical structure describes the legal arrangements of the logical elements within a certain type of document, such as memos, reports, letters, and so on. For example, a generic logical structure might specify that chapters can contain one or more sections, and sections can contain one or more paragraphs, but appendixes cannot contain chapters. A specific logical structure of a document is simply an instance of the generic logical structure for that type of document.

1.3.2 Document Layout

Document layout is defined as the manner in which document content elements (graphics, text, and images) are arranged on a page or series of pages. A compound document can be presented using a variety of page layout schemes. For example, the number of columns on a page and the

placement of page numbers and footnotes are all aspects of a document's layout.

More than any other aspect of document processing, the layout algorithm for a document differs between document processors and depends on the capabilities of the target device. Some terms associated with document layout are defined in Table 1–1.

Table 1–1 Layout Terminology

Term	Definition
Formatting	The process of fixing text in galleys; it involves breaking the stream of characters and floating frames into lines that fit within the assigned galleys. Formatting can also involve optimization of page layouts, the selection of appropriate page templates, and hyphenation decisions.
Galley	A rectangular guide, such as a column or footnote area. DDIF galleys are modeled by areas (usually rectangles) that are filled with text and relocatable illustrations during the formatting process.
Galley-based layout	In galley-based layout, characters and frames flow through a set of connected galleys and across pages instead of being fixed with respect to a coordinate system.
Generic layout	A set of rules that are used to determine the layout of a document or set of documents.
Page	A unit of display, such as a traditional sheet of paper, a video display, or a 35mm slide. A page is a discrete unit of content presented for viewing.
Specific layout	The layout of a particular document or document element.
Wrapping	The process of breaking a stream of characters into lines that fit within the assigned galleys.

The layout of a page is largely open to interpretation and preference. Page layout is generally guided by typographic conventions that have evolved throughout the history of printing. It is also influenced by the capabilities of the selected output device, as well as the capabilities of the formatter that is preparing the document for display.

Document layout can be generic or specific:

- The term **generic layout** describes a set of parameters and implicit (or explicit) methods used to determine the layout of document content. Generic layout typically specifies one or more page layouts and the linkages between them. For example, the first page of a chapter can contain a centered title and a half page of text, while the next page contains a full complement of text.
- **Specific layout** typically occurs in the final form of a document. That is, the layout of a document in its final form is referred to as the document's specific layout. However, specific layout can also occur in revisable documents where the content has been tied to a predetermined layout.

1.3.3 Logical Structure and Layout

The specific logical structure of a document often does not correspond to the layout. That is, the text content in the specific layout of the final form of a document can occur in a different order than it does in the specific logical structure of the revisable form. For example, a footnote would be stored at the first point of reference in a revisable document, but would appear at the bottom of the page in the final form.

The layout can also contain content that is not part of the logical structure. For example, page numbers inserted by a formatter are not part of the logical structure, but are part of the layout.

In revisable documents, content is stored and processed in the order corresponding to the logical structure. Final form documents are stored and processed in the order of the layout structure.

1.3.4 Structured and Unstructured Markup Systems

The extent to which the author of a document can control the arrangement of content on the page varies from system to system, from document to document, and often from place to place within a given document. Many formatters lay out text automatically, based on the galleys and the content at hand, while still allowing the author to insert formatting directives such as new line and new page. These kinds of directives are called **hard directives**, in the sense that they are permanent unless the user removes them. **Soft directives** are inserted by the software and are replaceable. (Soft directives are typically used by interactive editors to store pagination in order to reduce startup time for the next editing session.)

In markup languages that support structured documents (such as SGML), the layout process is governed by a style guide, which provides parameters to the formatter for each document type. The author of the document has little or no control over the layout process — each element of the document is formatted according to the corresponding set of parameters in the style guide. A given generic structure can have multiple styles, each specifying a different layout, so that the document can be formatted and displayed using different formats, perhaps for use with different display devices.

For example, a given system might support several style guides for manuals. The same chapter can be processed using these different style guides to produce 8½" by 11" output or 7" by 9" output, monospaced fonts or proportionally spaced fonts, and so on.

Some markup languages are not strictly structured, and allow the user to include layout directives in the document, in addition to or instead of a style guide.

1.3.5 Interactive and Batch Processing

An interactive editor may or may not support structure and a style guide; however, an interactive editor almost always allows the author to control the formatting process directly. Usually, an interactive editor provides some default format for a user-modifiable template that can be changed as needed. The formatting process must be fast; therefore, relatively little time can be spent on optimization. The user is compensated by the ability to interactively optimize the layout.

When the formatting process is not interactive, relatively ample processing time can be spent optimizing the layout of the document. For example, illustrations can be kept on the same page as the references, or on one of two pages when the document is intended for two-sided printing.

1.4 Separation of Layout from Content

Content laid out in galleys, on the basis of rules and parameters expressed as generic layout, can be laid out in a variety of ways — the number of columns, and the size and position of the columns, can be varied. Likewise, the line and page breaks can vary because of a variety of factors, such as the hyphenation decisions and the amount of white space optimization. Document content, such as a table, that is not laid out in a galley-based fashion generally cannot be rearranged without user interaction.

When specific layout instructions have been inserted into the document — that is, when the author has marked up the document for layout — then separation of the content from its layout involves removing or ignoring the specific markup and using only the generic markup of the document, or a default generic layout. For example, if a document that has been manually laid out in newspaper fashion is presented in a magazine, the specific layout is ignored and the generic layout model is used to format the document. If the same newspaper-formatted document is presented on a character-cell terminal, both the generic layout and specific layout are ignored and the content is laid out using a generic layout model suitable for character-cell terminals.

1.4.1 Replacement of Layout

In order to interchange documents, it is necessary to allow the layout of the document to be chosen by the software that encodes the document, and for the layout to be able to be changed by the software that receives the document.

The encoding application must be able to choose a layout scheme appropriate for its layout model — for example, an interactive editor cannot express the specific layout of a document using a generic layout model that is appropriate for a markup system. (A **markup system** typically consists of an integrated series of software processors that convert generically coded source files into formatted output.) Nor can markup systems express complex generic layout using the layout model supported by most interactive systems.

1.4 Separation of Layout from Content

There are two reasons why it might be necessary for the receiving application to replace the encoder's layout:

- The receiving application may not have a formatter capable of formatting the document.
- The receiving system might lack adequate display technology to support the encoder's selected layout.

For example, it is impossible to meaningfully display a multicolumn document set in 8-point type on an 80-column character-cell display. Instead, it is necessary to format the content in a way suited to the display device. Optionally, the receiving user might want to display and/or modify the document with an interactive editor that cannot support the CPU-intensive formatting that might have been specified in the document by the encoding application.

The sender's layout is replaced in the document itself only if the document is being modified. Otherwise, the new layout parameters are simply substituted during formatting and display. For example, if a document is mailed to a user with a character-cell terminal and a laser printer, the user can reformat the document so that it can be read on the terminal and then print the document in its original format on the laser printer to see the layout as the sender intended.

2 CDA Converter Architecture

The CDA Converter Architecture defines a methodology to simplify the conversion of compound documents. The CDA Converter Architecture is implemented through the following applications:

- The CDA Converter
- The DDIF Viewers
- The Converter front and back ends

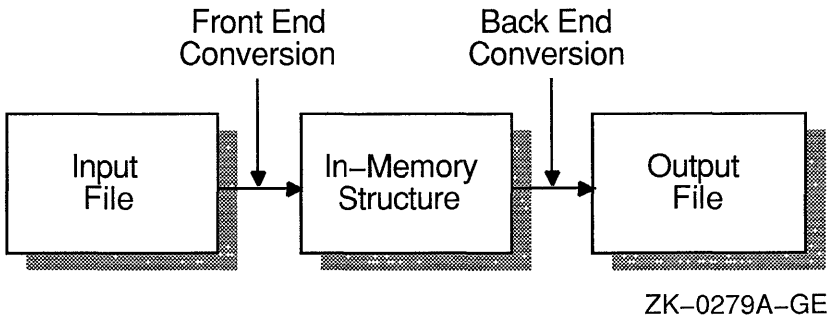
The following sections discuss each of these applications in more detail.

2.1 CDA Converter

The CDA Converter is an integral part of the Compound Document Architecture. It enables you to translate your compound document files to and from various file-encoding formats. The CDA Converter can be viewed as a "black box" that reads in an input file of the specified file-encoding format and converts it to an output file of the specified file-encoding format.

To accomplish this conversion, the CDA Converter uses the DIGITAL Document Interchange Format (DDIF) as the integral step in the conversion process. The converter reads the input file and translates it to a CDA in-memory format, and then translates this in-memory format to the specified output format. In other words, any input file-encoding format that is supported by the CDA Converter can be translated to a CDA in-memory format, and this in-memory format can subsequently be converted to any supported output file-encoding format. Figure 2–1 illustrates these basic stages of document conversion.

Figure 2–1 Stages of Document Conversion



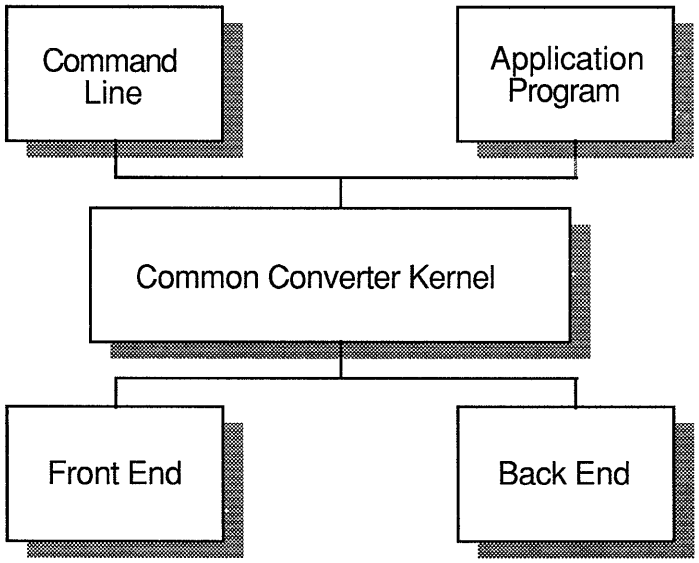
2.1.1 Components of a Converter

From the user's perspective, the converter is a "black box" that reads in the specified input file and converts it to the specified output file. For this single converter to be able to convert the wide variety of supported file-encoding formats, it actually comprises the following four parts:

- An interface (both a command line interface and an interface that is callable from within an application program)
- The CDA Converter Kernel that performs all the functions that must be completed for each conversion process, regardless of input and output formats
- A front end that converts a particular input format to the in-memory format
- A back end that converts the in-memory format to a particular output format

The relationship of the various converter components is shown in Figure 2–2.

Figure 2–2 Converter Components Diagram



When you invoke the converter, you always invoke the converter kernel first. This kernel performs the following functions:

- It performs all of the "generic" conversion functions that must be completed for every document conversion, regardless of input and output formats.

- It invokes the appropriate front end to translate the input file to the CDA in-memory format.
- It invokes the appropriate back end to translate the CDA in-memory format to an output file of the specified format.

The CDA Converter, therefore, actually consists of the CDA Converter Kernel, one front end for each supported input file-encoding format, and one back end for each supported output file-encoding format. The kernel translates the various file formats by calling the appropriate front end and back end to perform the requested conversion.

For example, if you have the CDA Converter Kernel, a DDIF front end, and an Analysis back end, you can invoke the converter to translate a DDIF-encoded input file to an Analysis-encoded output file. The common converter kernel invokes the DDIF front end and the Analysis back end to perform the requested conversion. In general, front ends and back ends are "paired." That is, if a file-encoding format is supported by a front end, it generally is also supported by a back end. However, this is not always the case. For example, the Analysis back end does not have a corresponding front end.

The front ends and back ends that are provided with the operating system are documented later in this chapter. Other available converters are documented in the appropriate application documentation sets. The interfaces to the CDA Converter are as follows:

- A DCL command line interface (CONVERT/DOCUMENT)
- A callable interface (the CONVERT routine) that is accessible from application programs

Each of these interfaces is discussed in the following sections. The supported input formats are discussed in Section 2.3 and the supported output formats are discussed in Section 2.4.

2.1.2 DCL CONVERT/DOCUMENT Command

The DCL CONVERT/DOCUMENT command invokes the conversion of a revisable format file to another revisable or final form file from the DCL command line. This command has the following format:

```
CONVERT/DOCUMENT[/OPTIONS=filespec]
    input-file[/FORMAT=fmt-name]  output-file[/FORMAT=fmt-name]
```

The /FORMAT qualifier enables you to specify the encoding formats of the input and output files. (DDIF is the default input and output format.) The format keywords for the supported input and output formats are listed in Table 2–1.

Table 2–1 Converter Format Keywords

Input Formats	Output Formats
DDIF	DDIF
TEXT	TEXT
N/A	PS
N/A	ANALYSIS

The /OPTIONS qualifier enables you to specify a file that contains options to be applied during the conversion of the file. Each line of the file specifies a format name that can contain upper- and lowercase alphabetic characters, digits, dollar signs, and underscores, optionally preceded by spaces and tabs, and terminated by any character other than those listed. Alphabetic case is not significant. The syntax and interpretation of the text that follows the format name are specified by the supplier of the front and back ends for the specified format. Multiple lines that specify the same format are permitted.

The following example illustrates a simple example of an options file that specifies options to be used when converting some file to a PostScript output file. The options disable word wrapping and page wrapping and specify the desired paper size.

```
ps word_wrap 0
ps page_wrap 0
ps paper_size legal
ps paper_orientation portrait
```

2.1.3 CONVERT Routine

The CONVERT routine invokes the conversion of a revisable format file to another revisable format or final form file from within an application program. This routine entry point has the following format:

```
CDA$CONVERT (function-code ,standard-item-list ,private-item-list
             ,converter-context)
```

The parameters to this routine are as follows:

- **Function-code** is a symbolic constant that identifies the function to be performed. Valid values for this argument are as follows:
  - — CDA$_START begins the conversion. This function code must be specified to begin a document conversion.
  - — CDA$_CONTINUE continues a conversion that was suspended. This function code can only be specified if a previous call to the CONVERT routine returned the value CDA$_SUSPEND. If CDA$_SUSPEND is returned by a call to the CONVERT routine, either CDA$_CONTINUE or CDA$_STOP must be specified so that resources locked by the conversion can be released.

— CDA$_STOP discontinues a conversion that was suspended. This function code can only be specified if the previous call to the CONVERT routine returned the value CDA$_SUSPEND. If CDA$_SUSPEND is returned by a call to the CONVERT routine, either CDA$_STOP or CDA$_CONTINUE must be specified so that resources locked by the conversion can be released.

- **Standard-item-list** is an item list that identifies the document source and destination and can also contain options to control processing. Valid code values for the items in the **standard-item-list** are as follows:

CDA$_INPUT_FORMAT
The address and length of a string that specifies the input document format.

CDA$_INPUT_FRONT_END_PROCEDURE
The address of the front end's main entry point: DDIF$READ_*format*. The item list length field must be 0. This item enables a caller to provide a front end that is part of the calling application rather than a separate image. If this item code is used, the CDA$_INPUT_FILE item can be used to pass any information (not necessarily a file specification) to the front end.

CDA$_INPUT_FILE
The address and length of a string that contains the file specification of the input document.

CDA$_INPUT_DEFAULT
The address and length of a string that specifies the default input file type. To simplify the porting of applications to other operating systems, the string should consist of only a file type in lowercase characters. If this parameter is omitted, a front end must supply an appropriate default file specification.

CDA$_INPUT_PROCEDURE
The address of a procedure to provide input. The item list length field must be 0. The input procedure must conform to the requirements for a user *get* routine. For more information on a user *get* routine, refer to the CONVERT routine description in Part II of this manual.

CDA$_INPUT_PROCEDURE_PARM
The address of a longword parameter to the input procedure. The item list length field must be 4.

CDA$_INPUT_ROOT_AGGREGATE
The address of a longword handle to a root aggregate that specifies an in-memory input document. The item list length field must be 4. The in-memory structure, except for the root aggregate itself, is erased by this operation. The root aggregate must specify standard memory allocation.

CDA$_OUTPUT_FORMAT
The address and length of a string that specifies the output document format.

CDA$_OUTPUT_BACK_END_PROCEDURE
The address of the back end's main entry point: DDIF$WRITE_*format*. The item list length field must be 0. This item enables a caller to provide a back end that is part of the calling application rather than a separate image. If this item code is used, the CDA$_OUTPUT_FILE item can be used to pass any information (not necessarily a file specification) to the back end.

CDA$_OUTPUT_FILE
The address and length of a string that contains the file specification of the output document.

CDA$_OUTPUT_DEFAULT
The address and length of a string that specifies the default output file type. To simplify the porting of applications to other operating systems, the string should consist of only a file type in lowercase characters. If this parameter is omitted, the back end must supply an appropriate default file specification.

CDA$_OUTPUT_PROCEDURE
The address of a procedure to receive output. The item list length field must be 0. The output procedure must conform to the requirements for a user *put* routine. For more information on a user *put* routine, refer to the CONVERT routine description in Part II of this manual.

CDA$_OUTPUT_PROCEDURE_PARM
The address of a longword parameter to the output procedure. The item list length field must be 4.

CDA$_OUTPUT_PROCEDURE_BUFFER
The address and length of the initial output buffer for the output procedure.

CDA$_OUTPUT_ROOT_AGGREGATE
The address of a longword handle to a root aggregate that receives an in-memory output document. The item list length field must be 4. The root aggregate must be empty, and must specify standard memory allocation.

CDA$_OPTIONS_FILE
The address and length of a string that contains the file specification of an options file specifying options to control processing. On VMS systems, the default file type is CDA$OPTIONS. Each line of the file specifies a format name, which may contain upper- and lowercase alphabetic characters, digits, dollar signs, and underscores, optionally preceded by spaces and tabs, and terminated by any character other than those listed. Alphabetic case is not significant. The syntax and interpretation of the text that follows the format name are specified by the supplier of the front and back ends for the specified format. Multiple lines that specify the same format are permitted.

- **Private-item-list** is a private item list that is passed directly to the back end invoked by the converter. The specification of this item list is the responsibility of the back end. Its purpose is direct two-way communication between the caller of the CONVERT routine and the back end.

- **Converter-context** is set to CDA$_START; this argument receives a value that must be specified as the converter context parameter when this routine is called with CDA$_CONTINUE or CDA$_STOP as the function code. This value is invalidated when the CONVERT routine returns a status other than CDA$_SUSPEND.

You can use this routine to invoke the converter from within an application program to perform file conversion.

2.2 DDIF Viewer

The DDIF Viewer is an application that enables you to view compound document files on a character cell terminal or workstation window. This Viewer works with the CDA Converter Architecture, so that a file of any input format supported by CDA can be viewed on a character cell terminal.

The Viewer works by converting an input file to the in-memory format used by the CDA Converter. This in-memory format is then formatted for output to the screen. In other words, the Viewer is a specific instance of the CDA Converter in which the output format is a screen display.

The interface to the DDIF Viewer is the DCL VIEW command. This command is discussed in the following section. The supported input formats for the DDIF viewer are described in Section 2.3.

2.2.1 DCL VIEW Command

The DCL VIEW command invokes the DDIF Viewer, which lets you view a compound document file on a character cell terminal or DECwindows display. Note that many of the text display attributes are not processed when displaying the document, because of the limitations of the viewing device.

The VIEW command has the following format:

```
VIEW input-file[/qualifiers]
```

The input file specifies the name of the file to be viewed. You cannot use wildcard characters in the file specification. The default input file-encoding format is DDIF, and the default file type is DDIF. Valid input file formats are DDIF and TEXT; these input formats are described in more detail in Section 2.3.

The qualifiers that you can specify to the view command are as follows:

- /FORMAT[=format-name]

  Specifies the format of the input file. The default format is DDIF. The appropriate front end must be available in SYS$LIBRARY for the specified **format-name**. The valid formats are DDIF and TEXT.

- /OUTPUT[=output-file-spec]

  Specifies a file that receives the text output. The default is /NOOUTPUT. If an output file specification is not specified, the output file specification defaults to input-file.LIS. If this qualifier is specified, the output of the VIEW command is not displayed on the screen, but is instead written to the specified file. Note that if you specify the /OUTPUT qualifier, you cannot also specify the /PAGE qualifier.

- /PAGE

  Controls the display of output, providing the same effect as the DCL TYPE/PAGE command when used on a non-DECwindows device. The default is /NOPAGE. The /PAGE qualifier has no effect when used with a DECwindows display because the scroll bars provide the same capability. Note that if you specify the /PAGE qualifier, you cannot also specify the /OUTPUT qualifier.

- /OPTIONS=file-spec

  Specifies a file that contains options to be applied during the conversion of the file to the CDA in-memory format. The default file type is DDIF$OPTIONS.

- /SELECT=select-list

  Allows the user to tailor the CDA Viewer output. The selection items you can specify are as follows:

[NO]GRAPHICS	Directs the viewer either to mark the location of graphics embedded in the DDIF file being processed by the DDIF viewer, or to ignore the graphics.
[NO]IMAGES	Directs the viewer either to mark the location of the images embedded in the DDIF file being processed by the DDIF viewer, or to ignore the images.
[NO]TEXT	Directs the viewer either to process the text contained in the DDIF file being processed, or to ignore the text.
ALL	Directs the viewer to process all information contained in the DDIF file being processed.
[NO]SOFT_DIRECTIVES	Directs the viewer either to process or ignore soft directives in the DDIF file being processed in order to format output. Soft directives specify such formatting commands as new line, new page, and tab.
[NO]AUTO_WRAP	Directs the viewer to perform word wrapping of any text that would exceed the right margin. NOAUTO_WRAP allows the text to exceed the margin.

[NO]X_DISPLAY	Directs the viewer to create a DECwindows widget to be used when viewing the file on a workstation display defined by the logical name DECW$DISPLAY. NOX_DISPLAY, the default, invokes the DDIF viewer. Note that X_DISPLAY cannot be specified if the /OUTPUT qualifier is also specified.

The default format is

```
/SELECT = (GRAPHICS, IMAGES, TEXT, SOFT_DIRECTIVES,
           AUTO_WRAP, NOX_DISPLAY)
```

2.3 Input Formats

The CDA Converter Architecture works by supplying a common converter kernel and front and back ends to support the various input and output formats. The following sections describe each supported front end, the data mapping between that input format and the in-memory format, any data loss that might occur during the conversion, and any other information specific to that front end.

2.3.1 DDIF Front End

The DDIF front end reads a file encoded in DDIF format and converts the information in the file to the CDA in-memory structure.

2.3.1.1 Data Mapping

Because the input file format is DDIF, the information in the file maps directly to the CDA in-memory structure.

2.3.1.2 Data Loss

The DDIF front end does not lose any data when converting a DDIF input file to the CDA in-memory structure. Again, this is because the input document type and the in-memory structure type are both DDIF.

2.3.1.3 External File References

When the DDIF front end encounters an external file reference that is specified in the document header of your DDIF input file, it passes the reference through to the CDA Converter Kernel.

2.3.1.4 Document Syntax Errors

If a document syntax error is encountered in the DDIF front end, that represents a fatal input processing error. The only way that this can occur is if the input document is invalid. If the DDIF front end does encounter a document syntax error, the conversion process is stopped and no further input processing is performed.

2.3.2 Text Front End

The Text front end reads a standard text (ISO Latin1) file and converts the information in the file to the CDA in-memory structure. If the text file was entered as a DEC Multinational Character Set file on a character cell terminal or terminal emulator, the following conversions occur:

Original Character	Converted Character
Concurrency sign	Diaeresis
Capital OE ligature	Multiplication sign
Capital Y with diaeresis	Capital Y with acute accent
Small oe ligature	Division sign
Small y with diaeresis	Y with acute accent

2.3.2.1 Data Mapping

When you invoke the converter for a Text input file, all of the text in the input file is mapped to DDIF text content. Line breaks and form feeds are mapped to DDIF directives. One or more contiguous blank lines are interpreted as end-of-paragraph markers.

2.3.2.2 Data Loss

The Text front end does not lose any data when converting a Text input file to the CDA in-memory structure. This is because no structure information is contained in a text file.

2.3.2.3 External File References

Text files do not contain external file references. Therefore, the Text front end does not evaluate external file references.

2.3.2.4 Document Syntax Errors

Because text files do not have any syntax defined, syntax errors cannot be encountered by the Text front end.

2.4 Output Formats

The following sections describe each back end supported by the CDA Converter Architecture, the data mapping between the in-memory format and the particular output format, any data loss that might occur during the conversion, and any other information specific to that back end.

2.4.1 DDIF Back End

The DDIF back end takes the CDA in-memory structure that has been converted from some input format, converts it to a DDIF output format, and writes the information to the specified DDIF output file.

2.4.1.1 Data Mapping

When you invoke the converter with the DDIF back end, the data mapping between the information in the CDA in-memory structure and the converted output file is one-to-one. This is because the in-memory structure type and the output document type are both DDIF.

2.4.1.2 Data Loss

The DDIF back end does not lose any data when converting a CDA in-memory structure to a DDIF output file. Again, this is because the in-memory structure type and the output document type are both DDIF.

2.4.2 Text Back End

The Text back end takes the CDA in-memory structure that has been converted from some input format, converts only the text content of the file, and writes the information to the specified text output file.

2.4.2.1 Data Mapping

When you invoke the converter for a text output file, all Latin1 text is written to the output text file.

2.4.2.2 Data Loss

When the Text back end is converting the in-memory structure to a text output file, all graphics, images, attributes, and formatting information are lost.

2.4.2.3 Processing Options

The text back end supports the following options:

ASCII_FALLBACK	This option causes the back end to output text in 7-bit ASCII. The fallback representation of the characters is described in the ANSI ASCII standard.
CONTENT_MESSAGES	This option causes the back end to put a message in the output file each time a nontext element is encountered in the in-memory CDA structures.

2.4.3 PostScript Back End

The PostScript back end takes the CDA in-memory structure that has been converted from some input format, converts the content of the file to PostScript-formatted information, and writes the information to the specified PostScript output file.

2.4.3.1 Data Mapping

When you invoke the converter for a PostScript output file, all document content is written to the output file.

2.4.3.2 Data Loss

When converting the in-memory structure to a PostScript output file, all document content is converted.

2.4.3.3 Processing Options

The PostScript back end supports the following processing options:

- PAPER_SIZE *paper-size*
- PAPER_HEIGHT *paper-height*
- PAPER_WIDTH *paper-width*
- PAPER_TOP_MARGIN *paper-top-margin*
- PAPER_BOTTOM_MARGIN *paper-bottom-margin*
- PAPER_LEFT_MARGIN *paper-left-margin*
- PAPER_RIGHT_MARGIN *paper-right-margin*
- PAPER_ORIENTATION *orientation*
- EIGHT_BIT_OUTPUT *eight-bit-output-state*
- OUTPUT_BUFFER_SIZE *output-buffer-size*
- SOFT_DIRECTIVES *soft-directives-state*
- WORD_WRAP *word-wrap-state*
- PAGE_WRAP *page-wrap-state*
- LAYOUT *layout-state*

The keyword is separated from its assigned value by one or more spaces or tabs. Note that, for all of the measurement options, the default unit of measure is inches (specified as "in"). Other supported units of measure are points (pts), centimeters (cm) and millimeters (mm).

The processing options are discussed individually in the following sections.

2.4.3.4 Paper Size Processing Option

The PAPER_SIZE *paper-size* option lets you specify the size of the paper to be used when formatting the resulting PostScript output file. Valid values for *paper-size* are as follows:

Keyword	Size
A0	841 x 1189 millimeters (33.13 x 46.85 inches)
A1	594 x 841 millimeters (23.40 x 33.13 inches)
A2	420 x 594 millimeters (16.55 x 23.40 inches)
A3	297 x 420 millimeters (11.70 x 16.55 inches)
A4	210 x 297 millimeters (8.27 x 11.70 inches)
A	8.5 x 11 inches
B	11 x 17 inches

Keyword	Size
C	17 x 22 inches
D	22 x 34 inches
E	34 x 44 inches
LEDGER	11 x 17 inches
LEGAL	8.5 x 14 inches
LETTER	8.5 x 11 inches
LP	13.7 x 11 inches
VT	8 x 5 inches

The A paper size (8.5 x 11 inches) is the default.

2.4.3.5 Paper Height Processing Option

The PAPER_HEIGHT *paper-height* processing option, in combination with the PAPER_WIDTH processing option, lets you specify a paper size other than one of the predefined values provided. The default paper height is 11 inches.

2.4.3.6 Paper Width Processing Option

The PAPER_WIDTH *paper-width* processing option, in combination with the PAPER_HEIGHT processing option, lets you specify a paper size other than one of the predefined sizes provided. The default paper width is 8.5 inches.

2.4.3.7 Top Margin Processing Option

The PAPER_TOP_MARGIN *top-margin* processing option lets you select the width of the margin provided at the top of the page. The default value is .25 inches.

2.4.3.8 Bottom Margin Processing Option

The PAPER_BOTTOM_MARGIN *bottom-margin* processing option lets you select the width of the margin provided at the bottom of the page. The default value is .25 inches.

2.4.3.9 Left Margin Processing Option

The PAPER_LEFT_MARGIN *left-margin* processing option lets you select the width of the margin provided on the left-hand side of the page. The default value is .25 inches.

2.4.3.10 Right Margin Processing Option

The PAPER_RIGHT_MARGIN *right-margin* processing option lets you select the width of the margin provided on the right-hand side of the page. The default value is .25 inches.

2.4.3.11 Paper Orientation Processing Option

The PAPER_ORIENTATION *orientation* processing option lets you select the paper orientation to be used in the output PostScript file. The valid values for the *orientation* argument are as follows:

Keyword	Meaning
PORTRAIT	The page is oriented so that the larger dimension is parallel to the vertical axis.
LANDSCAPE	The page is oriented so that the larger dimension is parallel to the horizontal axis.

The default is PORTRAIT.

2.4.3.12 Eight Bit Output Processing Option

The EIGHT_BIT_OUTPUT *eight-bit-output-state* processing option lets you select whether or not the PostScript back end should use 8-bit output. You can specify a value of either ON or OFF for the *eight-bit-output-state* argument. The default is ON.

2.4.3.13 Output Buffer Size Processing Option

The OUTPUT_BUFFER_SIZE *output-buffer-size* processing option lets you select the size of the output buffer. The value you specify must be within the following range:

$$64 \leq output - buffer - size \leq 256$$

The default is 132.

2.4.3.14 Soft Directives Processing Option

The SOFT_DIRECTIVES *soft-directives-state* processing option lets you select whether or not the PostScript back end processes soft directives in the DDIF file in order to format output. (Soft directives specify such formatting commands as new line, new page, and tab.) If the PostScript back end processes soft directives, the output file will look more like you intended.

You can specify a value of either ON or OFF for the *soft-directive-state* argument. The default is ON.

2.4.3.15 Word Wrap Processing Option

The WORD_WRAP *word-wrap-state* processing option lets you specify whether or not the PostScript back end performs word wrapping of any text that would exceed the right margin. You can specify a value of either ON or OFF for the *word-wrap-state* argument. The default is ON. If you specify OFF, the PostScript back end allows text to exceed the right margin.

2.4.3.16 Page Wrap Processing Option

The PAGE_WRAP *page-wrap-state* processing option lets you specify whether or not the PostScript back end performs page wrapping of any text that would exceed the bottom margin. You can specify a value of either ON or OFF for the *page-wrap-state* argument. The default is ON.

2.4.3.17 Layout Processing Option

The LAYOUT *layout-state* processing option lets you specify whether or not the PostScript back end processes the layout specified in the DDIF document. You can specify a value of either ON or OFF for the *layout-state* argument. The default is ON.

2.4.4 Analysis Back End

This back end produces an analysis of the CDA in-memory structure in the form of text output showing the named objects and values stored in the document. This is useful for debugging DDIF application programs.

The Analysis back end supports an /INHERITANCE processing option that specifies that the analysis is shown with attribute inheritance enabled. Inherited attributes are marked by "[default]" in the output.

3 Overview of DDIF

The DIGITAL Document Interchange Format (DDIF) describes the format used for the creation, storage, and interchange of revisable compound documents. In order to write a DDIF-conforming application using the CDA Toolkit routines, it is important to first understand some of the basic concepts of DDIF.

3.1 Document Content

Document content is defined as the information contained in the fundamental units of a document. Document content includes characters, lines, raster images, and so on. This is different from the attributes that are applied to content. **Attributes** specify how the information is presented; for example, attributes specify content characteristics such as font, line thickness, and color. Attributes can also specify how the information is stored; for example, image attributes control the storage of image content.

DDIF supports several types of document content:

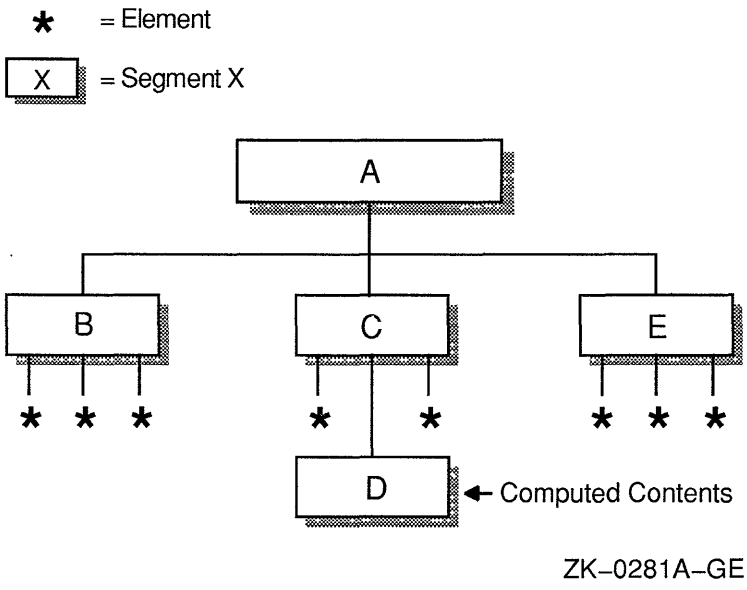
- **Text content** consists of text in ASCII and alternate character sets (including 16-bit text).
- **Graphics content** consists of primitives such as polylines and filled areas.
- **Image content** or **raster image content** consists of digitized images represented by actual values of monochrome, gray-scale, or color images.
- **Computed content** is document content (most often text content) that is calculated based on the current formatting state or other inclusion of external data. One example is a reference to the current page number, or to the page number on which a particular document element appears. The revisable form of the document describes the means by which the content is computed, while the final form of the document contains only the result of the computation (for example, the page number itself).

Document content can be either hard or soft. **Hard content** is entered by the creator of the document. **Soft content** is generated by software and is subject to recalculation when the document is revised. Page numbers used as cross-references are an example of soft text content. A chart generated from data to which the document is linked is an example of soft graphics content.

3.1.1 Document Hierarchy

DDIF represents a document as an ordered hierarchy of document segments. A **document segment**, or simply **segment**, is defined as a quantity of content that is set off from the surrounding data by a change in presentation or processing attributes. Each segment in a document contains document content, and can also contain nested segments. You can look at the hierarchy of segments as an inverted tree structure, in which case the segments are transmitted (or stored) from the top down and from left to right, simulating a depth-first traversal of the segment hierarchy. Content elements (the text, graphics, and images of the document) are displayed in this order.

Figure 3–1 Document Hierarchy



For example, the segments of the document illustrated in Figure 3–1 would be transmitted in the order A, B, C, D, and E. In this figure, the segment named A has B, C, and E as contents. Segments B and E are each shown as having three primitive content elements. Segment C also has three content elements, but one of these (D) is a nested segment. Segment D has no content; instead D contains computed content.

Example 3–1 illustrates the DDIF constructs (with the content omitted) representing the document shown in Figure 3–1.

Example 3–1 DDIF Document Sample

```
DDIF_DOCUMENT
{
 DDF_DESCRIPTOR
 {
  DSC_MAJOR_VERSION  1
  DSC_MINOR_VERSION  0
  DSC_PRODUCT_IDENTIFIER  "DDIF$"
  DSC_PRODUCT_NAME
  (
   ISO_LATIN1  "Hand-generated Standard DDIF Example"
  )
 }
 DDF_HEADER
 {
  DHD_VERSION
  (
   ISO_LATIN1  "V1.0"
  )
 }
 DDF_CONTENT
 {
  SEG_ID  "A"
  SEG_CONTENT
  {
   SEG_ID  "B"
  }
  {
   SEG_ID  "C"
   SEG_CONTENT
   {
    SEG_ID  "D"
   }
  }
  {
   SEG_ID  "E"
  }
 }
}
```

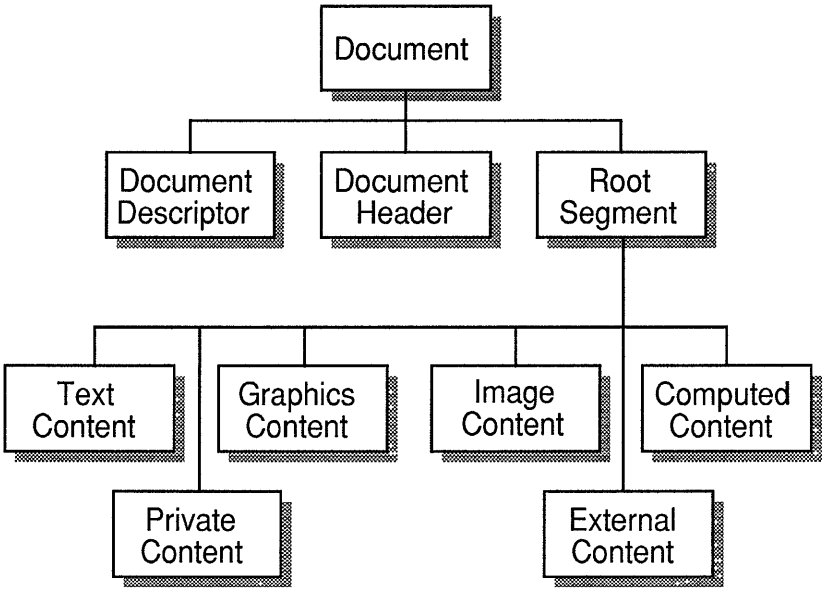
There are some structures (aggregates) that are required for every DDIF document; other constructs are optional, depending on the content of the document. An example of the hierarchical structure of a typical DDIF document is shown in Figure 3–2.

In Figure 3–2, the document is described in terms of a document root, a document descriptor, a document header, and the segments of document content that make up the document. Each of these pieces is described in the following sections.

3.1.2 Document Root

The document root identifies the document to an application that is processing the document. The encoding of the actual document root aggregate, as well as all other DDIF aggregates, is described in Chapter 6.

Figure 3–2 Typical DDIF Document



3.1.3 Document Descriptor

The document descriptor specifies information about the document, such as the DDIF version level used to encode the document and the software that created the document.

3.1.4 Document Header

The document header specifies certain information about the document as a whole. For example, the document header can specify the title of the document, the author, the version number of the document, a creation date, and any style guides to which the document conforms.

3.1.5 Root Segment

The content of a DDIF document is contained in a single segment called a root segment. This root segment contains zero or more content elements, including text, graphics, images, and nested segments. These standard content types express the basic units of meaning in a document and are described in the following sections.

3.1.5.1 Text Content

Text content consists of graphic characters and spaces from standard and private character sets. The presentation of the text is defined by text attributes that are specified using a segment attributes aggregate. Layout attributes describe the layout path to be used when the text is processed for presentation.

In addition, directives (such as new line and new page) are considered text content. These directives can either be hard (explicitly set by the user) or soft (inserted by the software that created the document for its own subsequent use). For example, if you specify a page break in a particular place in your document, that is a hard directive. If a text editor paginates your document during editing and saves these page breaks to reduce startup time, those are soft directives.

If a document is reformatted, the receiving or modifying application can ignore a soft directive. On the other hand, a hard directive cannot be ignored or removed, even if the document is reformatted. However, modifying applications enable the user to remove hard directives.

3.1.5.2 Graphics Content

Graphics content consists of such objects as polylines, cubic Bézier curves, arcs, fill areas, and paths that are created from a combination of the preceding objects. The presentation of graphics is defined by graphics attributes that are specified using a segment attributes aggregate. These graphics attributes describe such things as the line style, marker style, and fill patterns used for graphics content.

3.1.5.3 Image Content

Image content contains image data that is represented as a frame of data within a DDIF document. The origin of the frame is located at the lower left-hand corner of the frame. A frame can contain a single still image or a sequence of time-varying images with identical attributes. The presentation of these images is defined by image attributes that describe such presentation attributes as the pixel path and its aspect ratio, the brightness polarity of the image, and the physical format of the pixel grid in the image. Additionally, you can specify the attributes of the image component space, such as the number of data planes per pixel (and therefore per image) and the significance of the data planes.

3.1.5.4 Computed Content

Computed content is document content that is computed by a formatter or other document processor. Examples of computed content include section numbers, page numbers, and cross-references, in which the text content of the segment is generated by calculating the value of variables, such as the current page number. A segment whose content is computed must describe the method of its computation. It can additionally store its previously computed value, so that if none of its computation parameters have changed, the document formatter can eliminate the time required to recompute all of the computed content.

3.1.5.5 Restricted Content

Restricted content is provided in addition to the standard revisable content types. There are two types of restricted content: page description language (PDL) content and private content. In general, PDL content can only be displayed by the supporting devices, and is not suitable for revision. Private content indicates content that is restricted either to a particular document-processing implementation, or to a set of related implementations that support identical private encodings.

PDL content includes a stream of page description language in the content of the document; it is defined as an external data synax. Private content allows products or closely related product sets to include private markers, tags, and status information in document content.

3.1.5.6 Private Data

Private data is defined as document semantics that is restricted either to a particular document-processing implementation, or to a set of tightly coupled implementations that mutually support private encodings.

DDIF provides several instances where document processors can escape to private data, for example:

- In the header (for document-wide private indicators)
- In segment attributes (for hierarchical or inheritable data)
- As a content type (for content-like private data or markers)

Private data can be, for example, a marker in the document content that indicates the user's last editing position in the document, or a data element in the header of the document that indicates the menu setups or operation modes that the user had active at the time the document was written.

3.1.6 Relationships in Revisable Documents

In order to make a document revisable, DDIF defines different classes of relationships. These relationships are listed in Table 3–1.

Table 3–1 Relationships in Revisable Documents

Relationship	Meaning
Inheritance	This relationship defines a method for defaulting the attributes of content so that each segment of content does not need to specify all of its attributes. Instead, each segment inherits the attributes of the surrounding segment, and specifies only the difference between the attributes of its content and that of the surrounding content.

(continued on next page)

Table 3–1 (Cont.) Relationships in Revisable Documents

Relationship	Meaning
Generic attributes	This relationship defines attributes that can be applied to a number of segments, as opposed to being associated with a single segment.
Specific attributes	This relationship defines attributes that are associated only with a single segment of content. These types of attributes are deliberately limited to a specific segment of the document.
Generic type	This relationship defines a set of attributes and processing tags that define a type. Elements of the document can reference a defined type and become an "instance" of the type, thus inheriting the attributes and processing characteristics of the generic type.
Type reference	This actually represents a shorthand notation for the phrase "reference to generic type." When segments reference the same generic type, they inherit common attributes, and therefore take on common processing and presentation styles.
Generic content	This relationship defines document content that can be included in multiple places in the document. For example, a document containing several related illustrations might contain common graphics components that can be shared throughout the document or across a set of documents.
Content reference	This actually represents a shorthand notation for the phrase "reference to generic content." A content reference causes the generic content to be inserted into the final form when the document is formatted.
Variables	This relationship defines content that can be generated based on the values of variables, thereby ensuring that multiple elements of content are identical, have the same position, or can be modified by standard functions. For example, variables are used to indicate the numbering of list elements.
Style guide	This relationship defines a collection of generic types that are defined for use from a set of documents. A style guide takes the form of a document with definitions on the root segment, including type definitions, content definitions, font definitions, pattern definitions, line style definitions, and generic page descriptions. A document can contain only segmented content, and can make references to types in the externally defined style guide. Using different style guides makes it convenient to vary the style of a set of documents, or to vary the appearance of a given document. An example would be a style guide designed to match the capabilities of a target printer.

These revisable document relationships are referred to in the following sections.

3.1.6.1 Attribute Inheritance

As defined in Table 3–1, inheritance describes a method for defaulting the attributes of content so that each segment of content does not need to specify all of its attributes. In the document hierarchy, content attributes only affect the segment that declares or references them. In Figure 3–1, the attributes of C affect only the contents of C and its descendant, D. Segment D inherits all the attributes of C that D itself does not override, and also inherits the attributes of A that are not overridden by C. Any segment can therefore define the default attributes for its nested segments.

More specifically, the attributes that are inherited are those attributes that require some current value in order to make sense. For example, attributes such as line width, color, patterns, font definitions, and current font must always have some value; these attributes are therefore inherited if not explicitly declared. Attributes that are not inherited include segment identifiers, transformations, positions, and so on. These attributes are only specified through segment (generic) type inheritance or by direct specification, not through inclusion in the parent segment.

3.1.6.2 Generic Types

Any segment can define generic types which, in turn, can be referenced by nested segments. A **generic type** is defined as a set of attributes and processing tags that define a type. For example, you might create a generic type representing a footnote. Elements of the document can reference a generic type and become an instance of that type, inheriting the attributes and processing characteristics of the generic type. To continue the footnote example, whenever a footnote is required you can reference the generic footnote type to inherit the appropriate attributes for all footnotes throughout the document.

In Figure 3–1, segment C could define generic types that could be referenced from D, and segment A could define generic types that could be referenced from B, C, D, and E. Note that segments do not have to reference the generic type of the parent. However, if a segment wants to inherit the attributes associated with the generic type of its parent, it must explicitly reference that generic type.

3.1.6.3 Generic Content

In addition to generic types, a segment can also define generic content elements that can be used in any of its nested segments. **Generic content** is defined as document content that can be included in multiple places in the document. For example, a document that contains several related illustrations might contain common graphics components, which can be shared throughout the document or across a set of documents. Generic content can contain any of the DDIF content types, including nested segments. By using nested segments to define a generic content element, you can define complex content types in which content elements are differentiated by attributes.

For example, the user of a graphics editor might define a wheel consisting of a black tire, white spokes, and a gray wheel hub. This wheel could be defined in terms of graphics primitives and segmentation, and could then be referenced throughout all the diagrams of cars in the document. A change to the generic wheel would change the appearance of that wheel

throughout the entire document, because all specific instances of it are expanded from the single definition during the creation of the final-form document.

You can use references to a generic type when you are defining a generic content element. The definition of the generic type can be supplied either as part of the generic content definition, or it can be inherited through the parentage of the content reference. Note that a nested segment can redefine a generic type or a generic content element that is defined in a parent segment. In this case, the redefining segment and all its nested segments actually refer to the redefined generic element instead of to the original element.

3.1.6.4 References to Generic Types

When a segment references a generic type, it becomes a segment of that type and inherits any generic attributes associated with that type. These inherited attributes also apply to the descendants of the referencing segment. For example, in Figure 3–1, if A defines a generic type Q, and C references Q, then the generic attributes defined for Q take effect for C, and form the default attributes for D.

If an attribute is specified both in the referenced generic attributes and in the specific attributes, the specific attribute takes precedence. That is, specific attributes override generic attributes.

3.1.6.5 References to Generic Content

When you reference generic content, that content is inserted into the final form of the document when the document is formatted. This referenced generic content inherits the attributes of the segment in which the content reference occurs. However, segments within a generic content element can override the inherited attributes, just as they would if the generic content had occurred there directly. You can also use generic content to specify only some attributes, leaving others to be inherited from the segment in which they are referenced.

A content reference can specify a transformation to be applied to the generic content. All sizes and positions in generic content can be scaled, rotated, and translated.

3.1.7 Example of Document Content

Example 3–2 illustrates a small DDIF document and the various methods used to specify rendition attributes. This example is illustrated in the Analysis format — the format output by the Analysis Back End. In most cases, braces are used to enclose an aggregate, and parentheses are used to enclose an item that is encoded as an array.

Example 3–2 DDIF Document Attribute Inheritance

```
DDIF_DOCUMENT ❶
{
 DDF_DESCRIPTOR
 {
  DSC_MAJOR_VERSION  1 ❷
  DSC_MINOR_VERSION  0
  DSC_PRODUCT_IDENTIFIER  "DDIF$"
  DSC_PRODUCT_NAME ❸
  (
   ISO_LATIN1  "Hand-generated Standard DDIF Example"
  )
 }
 DDF_HEADER
 {
  DHD_VERSION ❹
  (
   ISO_LATIN1  "V0.1"
  )
 }
 DDF_CONTENT
 {
  SEG_SPECIFIC_ATTRIBUTES ❺
  {
   SGA_TYPE_DEFNS
   {
    TYD_LABEL  "BOLD" ❻
    TYD_ATTRIBUTES
    {
     SGA_TXT_RENDITION
     (
      RND_HIGHLIGHT
     )
    }
   }
   {
    TYD_LABEL  "UNDERLINED" ❼
    TYD_ATTRIBUTES
    {
     SGA_TXT_RENDITION
     (
      RND_UNDERLINE
     )
    }
   }
  } ❽
  SEG_CONTENT ❾
  {
   SEG_ID  "A" ❿
   SEG_SPECIFIC_ATTRIBUTES
   {
    SGA_TXT_RENDITION
    (
     RND_CROSS_OUT
    )
   }
```

(continued on next page)

Example 3–2 (Cont.) DDIF Document Attribute Inheritance

```
    SEG_CONTENT
    {
     TXT_CONTENT  "Text1" ⓫
    }
    {
     SEG_ID  "B" ⓬
     SEG_SEGMENT_TYPE  "BOLD"
     SEG_SPECIFIC_ATTRIBUTES
     {
      SGA_TXT_RENDITION
      (
       RND_DEFAULT
       RND_UNDERLINE
      )
     }
     SEG_CONTENT
     {
      TXT_CONTENT "Text2"
     }
    } ⓭
    {
     TXT_CONTENT  "Text3"
    }
   }
   {
    SEG_ID  "Goodness" ⓮
    SEG_CONTENT
    {
     SEG_SEGMENT_TYPE  "BOLD"
     SEG_CONTENT ⓯
     {
      TXT_CONTENT  "bold "
     }
     {
      SEG_SEGMENT_TYPE  "UNDERLINED"
      SEG_CONTENT
      {
       TXT_CONTENT  "bold underlined " ⓰
      }
     } ⓱
     {
      TXT_CONTENT  "bold again"
     }
    } ⓲
   } ⓳
  } ⓴
} ㉑
```

❶ This corresponds to the DDIF$_DDF_DESCRIPTOR item in the DDIF$_DDF aggregate. This item is encoded as the handle of a DDIF$_DSC aggregate which follows it in the structure.

❷ The DDIF$_DSC aggregate specifies the version number and product information of the product that created the document.

❸ The DDIF$_DSC_PRODUCT_NAME item in the DDIF$_DSC aggregate is encoded as an array of type character string. In this example, there is only one array value specified: ISO_LATIN1 "Hand-generated Standard DDIF Example".

❹ The items in the DDIF$_DHD aggregate are optional. In this example, only the version number is indicated. The DDIF$_DHD_VERSION item is encoded as an array of type character string. In this example, a single array item is specified: ISO_LATIN1 "V0.1".

❺ These segment-specific attributes are specified on the root segment of the document; hence, they can be referenced at any point in the document content. The attributes specified are "BOLD" (highlighted) and "UNDERLINED" (underlined).

❻ The bold attribute can be referenced using the label "BOLD". It is defined using a segment attributes (DDIF$_SGA) aggregate with the text rendition item (DDIF$_SGA_TXT_RENDITION) specified as DDIF$K_RND_HIGHLIGHT.

❼ The underlined attribute can be referenced using the label "UNDERLINED". It is defined using a segment attributes (DDIF$_SGA) aggregate with the text rendition item (DDIF$_SGA_TXT_RENDITION) specified as DDIF$K_RND_UNDERLINE.

❽ This right brace indicates the end of the definition of the segment-specific attributes.

❾ This marks the beginning of the content of the document. That is, this marks the DDIF$_SEG_CONTENT item of the root segment of the document. All of the document content is nested under this root segment.

❿ Segment "A" is the first segment nested under the root segment. This segment specifies a segment-specific attribute of crossed-out, so that all of its content will have a default attribute of crossed-out.

⓫ The first content aggregate in segment A is a text aggregate whose content is the string "Text1".

⓬ The second aggregate in segment A is a nested segment (B). This segment references the defined attribute "BOLD", and also specifies segment-specific attributes of default and underlined. The content of segment B is a text aggregate containing the string "Text2".

⓭ This right brace marks the end of the nested segment B. Segment A contains a third content aggregate — another text aggregate whose content is "Text3".

⓮ At this point, segment A and its content have been specified. This line marks the beginning of the segment entitled "Goodness". This segment, like segment A, is nested under the root segment.

⓯ Segment Goodness contains a nested segment that does not have a label but instead references the defined type BOLD. There are three aggregates nested under this aggregate: a text content aggregate, a nested segment, and another text content aggregate. The first text content aggregate contains the string "bold ". When this text is output, it will appear bolded.

⓰ The nested segment (nested under the segment referencing BOLD) references the defined type UNDERLINED. The content of this segment is a text aggregate containing the string "bold underlined ". When this text is output, it will appear bolded and underlined.

⓱ This right brace ends the nested segment (nested under the segment referencing BOLD). The last content aggregate of the segment referencing BOLD is a text content aggregate containing the string "bold again". When this text is output, it will appear bolded.

⓲ This right brace ends the segment referencing BOLD.

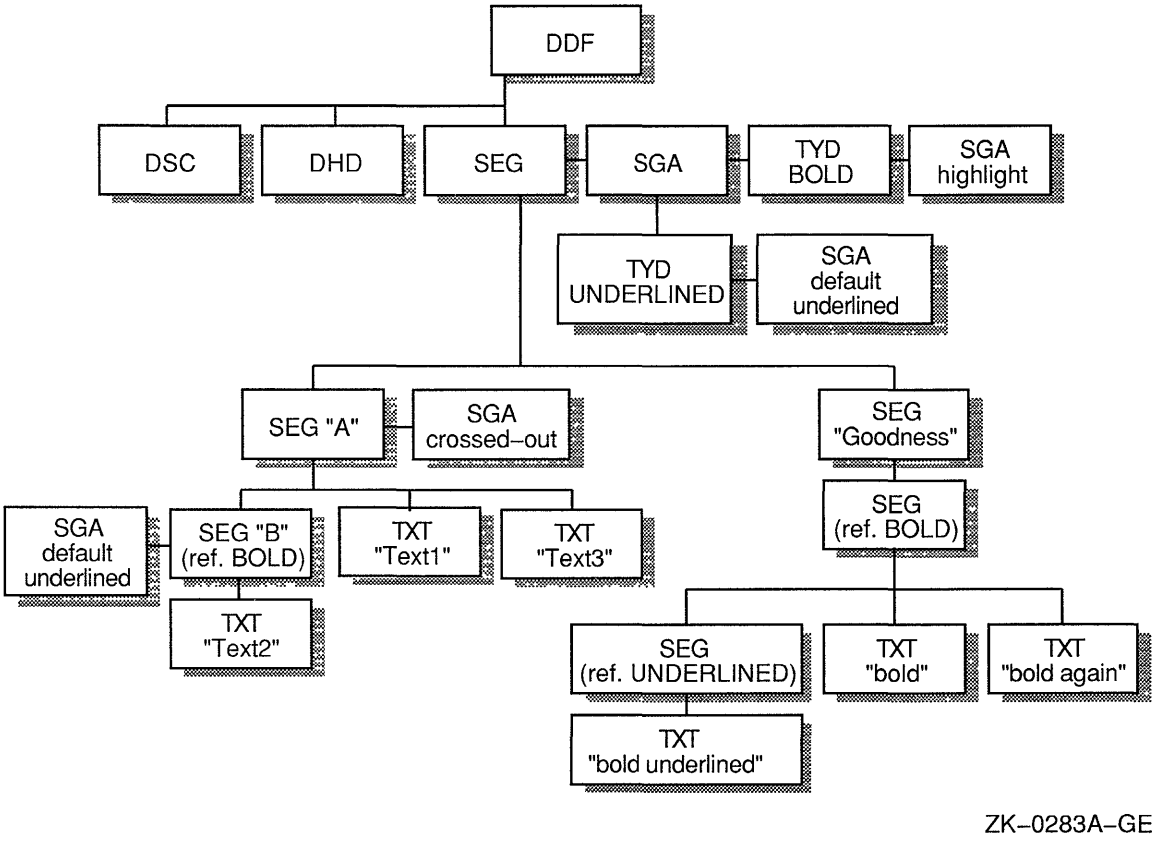
⓳ This right brace ends the segment Goodness.

⓴ This right brace ends the root segment.

㉑ This right brace ends the document.

Figure 3–3 illustrates the DDIF document described by the previous example.

Figure 3–3 Illustration of Inheritance Example Document



The renditions of the various text segments would be as follows:

- Text1's rendition list is { crossed-out }
- Text2's rendition list is { crossed-out, highlighted, default, underlined }
- Text3's rendition list is once again { crossed-out }
- The rendition of the "Goodness" segment would be as follows:
  bold bold underlined bold again

In general, to form the current rendition for any segment, the receiving software must process the list of renditions specified for the segments, modifying its current rendition state in response to each rendition. The "Goodness" segment illustrates the preferred method for using renditions: define types for the renditions and nest the segments on a per-rendition basis.

3.2 Document Layout

Document layout is defined as the manner in which document content elements (graphics, text, and images) are arranged on a page or series of pages.

The following sections summarize some of the typical approaches to layout in document processing systems.

To specify the generic layout of a document, you must define the layout parameters described in the following sections. Each of these parameters corresponds to a DDIF aggregate type; these aggregate types are described in Chapter 6. Note that generic layout descriptions can only be placed on the root segment of a document. Generic layout descriptions placed on segments other than the root segment are ignored. The same is true for specific layout descriptions.

3.2.1 Page Description

The page description provides a page model in the form of either a single page layout or a set of page layouts. If the page description is modeled by a set of page layouts, the description also specifies the conditions under which the different page layouts are used. In other words, if a page description is defined using a set of varying page layouts, you must also specify in the description which layout should be used under which conditions. For example, you might have a page description that consists of two actual page layouts: one for left-hand pages and one for right-hand pages.

3.2.2 Page Set

The page set specifies one or more pages, one of which is selected based on the current formatting state. Each page in the page set contains the following information:

- A pointer to a page in the list of page layouts

- The criteria for selection of that page

3.2.3 Page Layout

The page layout is used to describe a page, including such information as the page size, what galleys are on the page, and any content specific to that particular page. Note that this page layout syntax is used when you are specifying both generic and specific layout.

3.2.4 Galley

The galley layout specifies the shape and attributes of a single galley. A galley controls the flow of text along a series of parallel paths. These paths are determined by a formatter based on the following information:

- The outline of the galley
- The height of the characters on the lines
- Other layout parameters such as leading (**Leading** refers to the distance between lines of type.)

Galleys are relative to either a page frame defined by the page layout description, or to a floating frame. A galley will not be imaged when selected for filling with text, but rather in the normal sequence in which objects in the frame are imaged. A page frame and its contents are imaged when the first galley on the page is selected.

3.2.5 Implementation of Layout Separation

The content of a DDIF-encoded document is stored in logical order — the order in which the reader of the document would normally read it. The content of a document laid out in a newspaper style, for example, would be stored one article at a time, as opposed to having parts of the articles be interspersed with one another as they are in the page-ordered final form. The change in content order when the revisable form is converted to the final form is performed by the formatter.

The logically ordered content of the document is preceded by a specification of the generic and/or specific page descriptions. These are selected from within the content, or are simply used in the specified sequence.

Layout parameters and attributes are isolated from other types of attributes and from content, so a layout specification can be skipped without the formatter even knowing the syntax of that specification. It is therefore possible to display a DDIF document with complex galley-based layouts on character-cell devices even if the encoding application used an unrecognized layout specification.

The attributes that affect the layout of text (and floating frames) in the context of a galley-based layout are isolated in two individual attributes: wrap attributes and layout attributes.

3.2.5.1 Wrap Attributes

Wrap attributes let you specify parameters to control the process of wrapping text at the margin, as well as specifying hyphenation attributes and line format (centered, flush left, and so on). These attributes are applicable even if the galleys specified for the document are not used.

Wrap attributes do not determine where the line break occurs; they do not include margins or other dimensional parameters. Because the wrap attributes are independent of the dimensions, they can be applied when layout dimensions are discarded. For example, when an application is presenting a compound document on a character-cell device, the hyphenation limits and the line format still convey meaningful information.

3.2.5.2 Layout Attributes

Layout attributes, unlike the wrap attributes, include physical dimensions that require a layout template as a frame of reference. Examples of such dimensions include margins, indents, and tab stops. A formatter that is not using the specified page layout templates cannot use the layout attributes, and should replace them with attributes appropriate for the page descriptions actually being used.

3.2.6 Content Streams in Layout

A given galley on a page accepts content only from certain streams. For example, footnote galleys accept content only from the footnote stream. Thus, while the footnote content is logically embedded within the content of the paragraph that references it, it appears in the galley at the bottom of the page, or even at the end of the chapter. Therefore, DDIF provides a method to tag content elements by stream.

Once a content element is tagged by stream, a formatter can be instructed to include only certain streams in the document layout, so that variants on a document can be produced at the user's option. For example, comments on the document can be left out of production runs, while being included in special review drafts.

Each stream type is identified by a label or tag. The types of streams that exist for a document include:

- Document body content stream ($DB)
- Table of contents stream ($TOC)
- Index content stream ($IX)
- Footnote stream ($FN)
- Margin note stream ($MN)
- End note stream ($EN)

Elements that appear in both the table of contents and the document body (for example, section heads) should be tagged for appearance in two streams — the document body and the table of contents. When a revisable document includes a table of contents, the table of contents is contained in a segment with a computed content attribute that specifies a table-of-contents generating function. The content of that segment does not have the table of content stream ($TOC) tag, but rather the document body ($DB) tag because it is part of the body of that document. If the table of contents is regenerated, the contents of the table of contents segment are discarded and regenerated from the $TOC-tagged elements in the document. The same situation applies to indexes, except that index elements often do not appear in the body of the document and therefore are not part of that stream.

4 Overview of the CDA Toolkit

The CDA Toolkit routines enable you to write a DDIF-conforming application without having to know the specifics of the DIGITAL Document Interchange Format. This chapter provides an overview of the capabilities of the CDA Toolkit, as well as a description of the terminology associated with the Toolkit.

4.1 CDA Toolkit Routines Terminology

The definitions discussed in Chapter 3 are used in reference to the DIGITAL Document Interchange Format. In the discussion of the CDA Toolkit routines, the terminology listed in Table 4–1 is also used.

Table 4–1 Routines Terminology

Term	Definition
Aggregate	An in-memory structure that is used to pass compound document data between the application and the Toolkit routines. An aggregate corresponds to a manageable unit of the compound document. Aggregates are typed and self-describing; the type of an aggregate is indicated by a symbolic constant. An aggregate can be a member of an aggregate sequence, which can be traversed from beginning to end. Aggregates are defined for such objects as a document root, document descriptor, document header, document segments, text content, and so on.
Attribute	A presentation or processing characteristic.
Document	An entire hierarchical structure in memory, created by the CDA Toolkit routines.
Handle	The identifier of an aggregate.
Item	The identifier of a specific unit of information stored in an aggregate. The handle of an item is a symbolic constant defined in the file DDIF$DEF.SDL.
Root aggregate	An aggregate that represents the root of the in-memory document hierarchy. It also contains context private to the Toolkit routines. The type of the root aggregate is DDIF$_DDF.
Segment	A quantity of content that is set off from surrounding data by a change in presentation or processing attributes.
Sequence	A linked series of aggregates.
Stream	An access path by which encoded compound document data is transferred.

The CDA Toolkit routines are designed to simplify the creation and manipulation of compound document data. The routines provided by the Toolkit perform the following operations:

- File management
- Stream management
- Aggregate management
- Document conversion
- Item access
- Front end activation

The CDA Toolkit routines are discussed in the following sections.

4.2 File Management

The CDA Toolkit provides several routines to implement file management. To open or create a compound document file, the CDA Toolkit provides two routines: the OPEN FILE routine opens an existing compound document file for input, and the CREATE FILE routine creates a new compound document file for output. Each of these routines is discussed in the following paragraphs.

The OPEN FILE routine opens an existing compound document file for input and confirms that the contents of the file are valid compound document data. Once the file is opened, the OPEN FILE routine returns the file and stream handles (identifiers) for the opened file; these handles must be used in all subsequent operations on the file or stream. The OPEN FILE routine also creates a document root aggregate and returns the root aggregate handle, which must be used in all subsequent operations on that document root aggregate. You therefore do not have to invoke the CREATE ROOT AGGREGATE routine after calling the OPEN FILE routine.

The CREATE FILE routine creates a new compound document file for output and prepares it to receive data from a compound document stream. Once the new file is created, the CREATE FILE routine returns the file and stream handles for the new document; these handles must be used in all subsequent operations on the file or stream. Because the CREATE FILE routine creates a new file, you must create a document root aggregate (by a call to the CREATE ROOT AGGREGATE routine) prior to a call to CREATE FILE; this root aggregate handle must be passed to the CREATE FILE routine to identify the document being created. This root aggregate handle must be used in all subsequent operations on that document root aggregate.

The CLOSE FILE routine closes the currently open compound document stream and file. In the case of an output file, the CLOSE FILE routine writes any remaining buffered data to the output stream before closing the compound document file.

The CDA Toolkit also provides several routines to simplify text file management. On VMS systems, a standard text file has variable-length records and CR carriage control.

The OPEN TEXT FILE routine opens a standard text file for input. On VMS systems, a standard text file is any RMS file with variable-length records and carriage return record attributes. You can then use the READ TEXT FILE routine to read a line from a standard text file. On VMS systems, the line that is read is the next RMS record.

The CREATE TEXT FILE routine creates a standard text file for output. You can then use the WRITE TEXT FILE routine to write a line to this standard text file. On VMS systems, the line becomes an RMS record.

The CLOSE TEXT FILE routine closes a standard text file. The handle of the text file is invalid after a call to this routine.

4.3 Stream Management

Stream management routines are provided for application programs that require additional control (not provided by the file management routines) over the source or destination of a compound document stream. For example, the stream management routines can be used when the source or destination is not necessarily a file that resides on the host system.

The stream management routines are as follows:

- The OPEN STREAM routine opens a compound document stream for input.
- The CREATE STREAM routine creates a compound document stream for output.
- The CLOSE STREAM routine closes the specified stream and invalidates the stream handle. In the case of an output stream, this routine writes any buffered data before closing the stream.
- The FLUSH STREAM routine ensures that the data has been physically transferred to the receiving medium by writing any buffered data to the specified output stream.

An in-memory document exists independently of a stream. Once you create an in-memory document, you can populate it either by reading compound document data from a stream or by creating the aggregates that make up the document. Once you populate the in-memory document, you can write its data to a stream. The number of documents that can exist simultaneously in memory is limited only by the amount of memory available.

4.4 Root Aggregate Management

The first aggregate that must be created for a compound document is the document root aggregate. If you are reading a compound document file, you do not have to create a root aggregate explicitly because, when you open a compound document file, the OPEN FILE routine automatically creates a root aggregate and returns the root aggregate handle. However, if you are opening a file for output using the CREATE FILE routine, you must explicitly create an aggregate.

If you are reading or writing a file that is not a compound document, you must explicitly create a document root aggregate before creating any other aggregates. The document root aggregate is used to identify the compound document and to begin the tree structure that contains all of the aggregates that make up that compound document.

The CREATE ROOT AGGREGATE routine creates a document root aggregate and returns its handle. This root aggregate handle must be used to identify the compound document in all subsequent operations on that compound document. The CREATE ROOT AGGREGATE routine accepts a **processing-options** parameter that you can use to specify processing options that remain in effect for the life of the document. Processing options that you can specify include inheriting attributes, retaining definitions, evaluating contents, and discarding information.

The DELETE ROOT AGGREGATE routine deletes a document root aggregate and all of its substructure. It does not, however, delete aggregates that were created with the root but not connected to it physically in the tree. The handle of the root aggregate, as well as the handle of any aggregate linked to the root aggregate either directly or indirectly, is invalid after a call to this routine.

The CDA Toolkit provides routines to translate a root aggregate to a DIGITAL Document Interchange Syntax (DDIS) type object identifier, and to translate an object identifier to a root aggregate. The AGGREGATE TYPE TO OBJECT ID routine translates a root aggregate type to an object identifier; the OBJECT ID TO AGGREGATE TYPE routine translates a DDIS object identifier to a root aggregate.

4.5 Aggregate Management

Once you have created the document root aggregate, you should use the CREATE AGGREGATE routine to create aggregates of the following types:

- A document header aggregate (type DDIF$_DHD)
- A document descriptor aggregate (type DDIF$_DSC)
- A parent segment aggregate (type DDIF$_SEG)

These aggregates must be present in every compound document.

The CREATE AGGREGATE routine creates a new aggregate that contains empty items. Once an aggregate is created, it can be filled or populated using the STORE ITEM routine. For more information on the STORE ITEM routine, see Section 4.6.

Along with the parent or root segment aggregate, you should also create any generic type definition aggregates that you want to be available to the entire document content. An example of generic types is illustrated in Chapter 3. Once all of these aggregates are created, you can begin creating the aggregates that contain the actual document content.

Note: If an aggregate (A) contains an item whose value is the handle of another aggregate (B), then the latter aggregate (B) is called a subaggregate. That is, a subaggregate is connected to the document hierarchy by storing its handle as an item in another

aggregate. For example, the generic layout item in the segment aggregate contains the handle of the generic layout aggregate; the generic layout aggregate is therefore a subaggregate.

The COPY AGGREGATE routine creates a copy of the specified aggregate. This aggregate copy is assigned a unique aggregate handle and becomes part of the document associated with the specified root aggregate. If the specified aggregate is part of a sequence, only the aggregate specified, rather than the entire sequence, is copied.

The DELETE AGGREGATE routine deletes an aggregate and all of its substructure. If the aggregate being deleted is part of a sequence, it is first cut from the sequence before being deleted. The aggregate handle of the deleted aggregate, and the aggregate handles of any aggregates linked to the deleted aggregate either directly or indirectly, are invalid after a call to this routine.

The PRUNE AGGREGATE routine removes the next sequential document content aggregate from an existing in-memory document and returns the handle and type of the removed aggregate. This routine should be used by the *get-aggregate* procedure that is invoked when a front end builds an entire compound document in memory before returning its content. For more information on writing converter front and back ends, see Chapter 5.

The CDA Toolkit also provides aggregate-structuring routines that are used to scan and modify an aggregate sequence. These aggregate-structuring routines provide the following capabilities:

- Place a new aggregate in a sequence
- Delete an aggregate from a sequence
- Scan all the aggregates in a sequence

The INSERT AGGREGATE routine inserts an aggregate into a sequence. The position at which the aggregate is to be inserted is specified by indicating the aggregate handle of the preceding aggregate in the sequence. If the aggregate being inserted is the first aggregate in its own sequence, then the entire sequence is inserted after the specified aggregate and before the subsequent aggregate in the original sequence. If the aggregate being inserted is part of a sequence, but is not the first aggregate in the sequence, an error is returned.

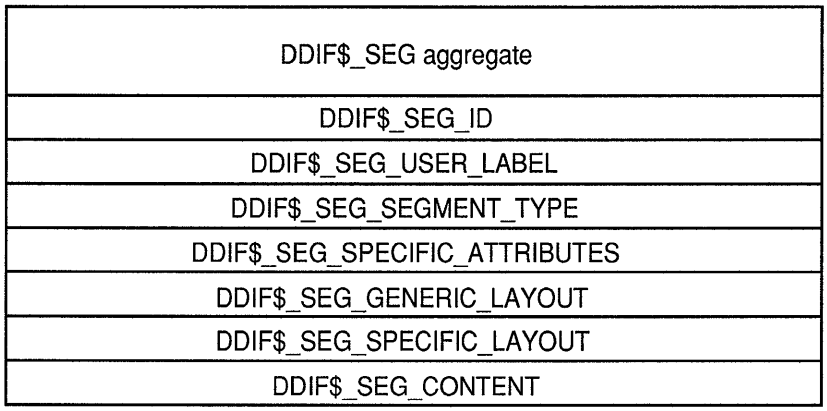
The REMOVE AGGREGATE routine removes an aggregate that is part of a sequence from that sequence. If the specified aggregate is not part of a sequence, no operation is performed.

The NEXT AGGREGATE routine locates the next aggregate in a sequence.

4.6 Item Access

A compound document aggregate is a type of record or data structure. Each aggregate contains certain items that identify that particular instance of the aggregate. The items that are contained in each aggregate are described in Chapter 6.

Figure 4–1 Document Segment Aggregate



ZK–0284A–GE

For example, Figure 4–1 illustrates the structure of a segment aggregate (type DDIF$_SEG) with all of its items present.

The item access routines are used to write, delete, and locate the items in an aggregate. Each item in an aggregate can be read, modified, or erased individually. Aggregate items can be of a number of data types, or can contain handles of other aggregates, so that you can read an item that contains an aggregate handle and then use that handle to read or modify the subaggregate. Table 4–2 lists the possible item data types and their meanings.

Table 4–2 Item Data Types

Data Type	Meaning
Byte	A byte. The length of the item buffer is always 1.
Boolean	A byte representing a Boolean value. The length of the item buffer is always 1. If the low bit of the value is set, the value is true. If the low bit is clear, the value is false.
Word	A word. The length of the item buffer is always 2.
Longword	A longword bit-encoded structure. The bits are interpreted according to a defined structure. The length of the item buffer is always 4.
Integer	A longword integer. The length of the item buffer is always 4.
Enumeration	A longword integer. The allowed values of the integer are defined by symbolic constants. The length of the item buffer is always 4.

(continued on next page)

Table 4–2 (Cont.) Item Data Types

Data Type	Meaning
String	A string of bytes. The length of the string is specified in bytes.
Bit string	A string of bits. The length of the item buffer is expressed in bits rather than bytes.
DDIF$_xyz	A longword aggregate handle to an aggregate of the specified type. The length of the item buffer is always 4.
Character string	A string of bytes in a particular character set (for example, ISO Latin1). The *add-info* parameter receives the character set designator. The symbolic constants for the character set designators are defined in module CDA$DEF.SDL.
Variable	The data type of the item is determined by a reference to the value of the preceding enumeration item. A variable type is always preceded by an enumeration item that specifies the data type of the variable item.
Content	A shorthand for any one of the following: • DDIF$_SEG • DDIF$_TXT • DDIF$_GTX • DDIF$_HRD • DDIF$_SFT • DDIF$_HRV • DDIF$_SFV • DDIF$_BEZ • DDIF$_LIN • DDIF$_ARC • DDIF$_FAS • DDIF$_IMG • DDIF$_CRF • DDIF$_EXT • DDIF$_PVT • DDIF$_GLY
Measurement enumeration	An enumeration that specifies the data type of an item of DDIF type Measurement, which is encoded as an integer or string. A DDIF Measurement type can either specify a specific number of measurement units, or it can specify the number of measurement units given by the value of the referenced variable.

(continued on next page)

Table 4–2 (Cont.) Item Data Types

Data Type	Meaning
AngleRef enumeration	An enumeration that specifies the data type of an item of DDIF type AngleRef, which is encoded as a floating point or string. A DDIF AngleRef type can either specify a constant angle value, measured in degrees, or it can specify an angle value derived from the value of the referenced variable.
Expression enumeration	An enumeration that specifies the data type of an item of DDIF type Expression, which is encoded as an integer or string. A DDIF Expression type can either specify a constant expression value, or it can specify an expression value derived from the value of the referenced variable.
Single-precision floating	A VAX F_floating point value. The length of the buffer is always 4.
Object identifier	Two or more longwords that specify the value of the DDIS type OBJECT IDENTIFIER. (DDIS is the DIGITAL Document Interchange Syntax.) Each longword specifies a single component of the object handle. The length of the item buffer is expressed in bytes.
Binary relative time	A binary relative time whose buffer length is always 16.
Document	A longword aggregate handle that contains the root aggregate handle of a subdocument.
DDIS encoding	A string of bytes. The length of the string is specified in bytes.
Item change list	A vector of longwords in which each longword contains the item code of an item in a segment attribute aggregate (DDIF$_SGA).
String with *add-info*	A string of bytes that represents the value of the DDIF type Tag, where standard tag values have been defined. As a service to the application, the CDA Toolkit provides encoding and decoding services for the standard tags.

In addition, in the descriptions of the encodings of aggregate items, the notation "Array of" indicates that the CDA Toolkit stores the item values in an array. To fill this array, you must specify one item value at a time, along with an aggregate index value. The initial aggregate index value is 0; you must increment this aggregate index each time you write an item value into the item encoded as an array.

The notation "Sequence of" indicates that the value of an item can be an aggregate sequence. A sequence is a linked list of aggregates of the specified type. The value of the aggregate item that is encoded as a sequence is actually the handle of an aggregate of the same type, and that aggregate can also contain an item that is actually the handle of another aggregate of that type.

The LOCATE ITEM routine determines the address of an item within an aggregate. The STORE ITEM routine writes the contents of an item in an aggregate, thereby filling in the contents of the aggregate. Each DDIF aggregate and the items it contains is specified in Chapter 6. If the aggregate item is indexed, the index specified must not exceed one more than the number of existing items. If the item is of data type Variable, the value of the item that determines the data type must have been previously established.

Note that the STORE ITEM routine erases the previous item value, unless the item is "aggregate-valued" and not empty. (An *aggregate-valued* item is one in which the value of the aggregate is actually the handle of another aggregate.) In the case of an item that is aggregate-valued and not empty, calling the STORE ITEM routine causes the specified aggregate to be inserted in sequence before the existing aggregate. If the specified aggregate is the beginning of a sequence, the entire sequence is inserted before the existing aggregate. If the specified aggregate is part of a sequence but is not the first aggregate in the sequence, an error is returned.

The ERASE ITEM routine erases (sets to empty) the contents of an item within an aggregate. If you erase an item that is indexed, the index of each subsequent item is decreased by 1. The GET ARRAY SIZE routine determines the number of elements present in an array-valued aggregate item.

4.7 Document Conversion

There are two different methods that you can use to perform document conversion:

- You can read or write an entire document to or from a stream.
- You can read or write single aggregates to or from a stream.

Both of these methods accomplish the same end result; in general, the second (incremental) method should be used when the characteristics of the output document format do not require that the entire input document be available in memory in order to complete the conversion.

The CDA Toolkit supplies corresponding routines to simplify both document conversion methods. These routines are as follows:

Document Method	Aggregate Method	Function
CONVERT DOCUMENT	CONVERT AGGREGATE	Used by the back end to request the appropriate information from the front end
GET DOCUMENT	GET AGGREGATE	Used by the front end to read the appropriate data from a compound document file
PUT DOCUMENT	PUT AGGREGATE	Used by a back end to write the converted information to a compound document file
PRUNE POSITION	CONVERT POSITION	Used by a viewer back end to determine current location in the document

Each of these routines is discussed in more detail in the following sections.

4.7.1 Document Transfer

The CONVERT DOCUMENT routine is used by a back end to invoke a front end. The front end reads in the entire document so that, on return from this routine, the entire compound document is present in memory in the form of aggregates linked from the document root aggregate. The CONVERT DOCUMENT routine accepts a **front-end-handle** argument. This argument specifies the front end that will perform the input processing.

If the input document is DDIF-encoded, the front end can invoke the GET DOCUMENT routine to read the entire compound document from the specified stream, create the appropriate aggregates, and insert them in the hierarchical structure. (Note that a root aggregate must exist before you call the GET DOCUMENT routine.) When a call to this routine is completed, the entire document is present in aggregates linked from the document root aggregate.

If the input document is encoded in some format other than DDIF, the appropriate front end must perform its own processing to create the necessary aggregates and insert them in the document structure in memory. For more information on writing a front end, see Chapter 5.

The PRUNE POSITION routine returns the current position in and total size of an in-memory document. This routine is used by viewer back ends that provide scroll bar support to indicate the current position in the document.

The PUT DOCUMENT routine is used by a back end that is writing DDIF-encoded output. This routine traverses an existing complete document hierarchy and writes the information in the aggregates to the compound document output stream. The document is unchanged by this operation. If, after a call to this routine, you no longer require the in-memory document, you should call the DELETE ROOT AGGREGATE routine to destroy the document and all its substructure.

If the selected output format is not DDIF, the appropriate back end must perform its own processing to convert the in-memory document to the specified format and write the information to the output stream. For more information on writing a back end, see Chapter 5.

4.7.2 Aggregate Transfer

The aggregate transfer routines let you incrementally process a compound document. Your application regains control after each header or content aggregate and its subaggregates are read or written.

The CONVERT AGGREGATE routine is used by a back end to invoke a front end. The front end reads a single aggregate and its subaggregates at one time so that, on return from this routine, a single aggregate and its subaggregates, not the entire document, are present in memory at any given time. The CONVERT AGGREGATE routine reads the next

aggregate from the specified front end module; this returned aggregate is not part of a sequence. The CONVERT AGGREGATE routine accepts a **front-end-handle** argument; this argument specifies the front end that will perform the input processing.

If the input document is DDIF-encoded, the front end can invoke the GET AGGREGATE routine to read the next aggregate from the document. (Note that the aggregate returned is not part of a sequence.) If the input document is encoded in some format other than DDIF, the appropriate front end must perform its own processing to create the necessary aggregate and fill in the appropriate items. For more information on writing a front end, see Chapter 5.

The GET AGGREGATE routine reads the aggregates in a document in a hierarchical fashion. That is, whenever GET AGGREGATE encounters a segment, it descends to the next level of hierarchy and reads the contents of that segment before reading the remaining contents of the parent segment. The GET AGGREGATE routine only returns to the parent segment's level of hierarchy when it encounters a DDIF$_EOS (end of segment) aggregate to indicate that the nested segment is completed. These rules can be generalized as follows:

- If the aggregate being read is a content aggregate, the aggregate is simply returned and the next aggregate returned is the next aggregate in the segment.
- If the aggregate being read is a segment aggregate (DDIF$_SEG), the content nested in the segment is returned, using these same ordering rules, followed by a dummy DDIF$_EOS (end of segment) aggregate to indicate the end of the nested segment. Once the nested segment and its content have been returned and the end of the segment has been indicated, the next aggregate read is the next aggregate in the (current) segment.

Note: All segments must be completed by a DDIF$_EOS aggregate.

The CONVERT POSITION routine returns the current position in and total size of the input stream being processed. This routine is used by viewer back ends providing scroll bar support that indicates the current position of the data being viewed in the document.

The PUT AGGREGATE routine writes one or more aggregates to the specified compound document stream. (If the aggregate being written is part of a sequence, the entire sequence is written.) The aggregate that is written is unchanged by this operation. If the aggregate is no longer required after you call the PUT AGGREGATE routine, you should call the DELETE AGGREGATE routine to destroy the aggregate.

If the selected output format is not DDIF, the appropriate back end must perform its own processing to convert the aggregates in the in-memory document to the specified format and write each aggregate and its subaggregates to the output stream. For more information on writing a back end, see Chapter 5.

When you are incrementally writing a document, you must invoke the ENTER SCOPE and LEAVE SCOPE routines to properly structure the output stream as aggregates are output. The ENTER SCOPE routine opens a document scope for incremental writing. The **scope-code** parameter to this routine lets you specify the type of scope being opened:

Document scope	DDIF$K_DOCUMENT_SCOPE
Content scope	DDIF$K_CONTENT_SCOPE
Segment scope	DDIF$K_SEGMENT_SCOPE

The LEAVE SCOPE routine completes a scope that was being incrementally written.

If you are incrementally writing a document (that is, writing the document one aggregate at a time), you should perform the following steps to ensure that the output stream is properly structured:

1. Call the ENTER SCOPE routine, specifying **scope-code** as DDIF$K_DOCUMENT_SCOPE.
2. Write an aggregate of type DDIF$_DSC.
3. Write an aggregate of type DDIF$_DHD.
4. Call the ENTER SCOPE routine, specifying **scope-code** as DDIF$K_CONTENT_SCOPE.
5. Write a root segment of type DDIF$_SEG. The root segment is a top level segment that contains the document content. This document content can consist of content aggregates as well as nested segments. If the document contains only one segment, that segment is the root segment and it contains all of the document content. If the document contains multiple segments, they must be nested within a root segment.

   You can use either of the methods outlined in the following steps to create the root segment. Because the first method requires that the entire segment be completed before calling the PUT AGGREGATE routine, once you select that method you must continue to use that method while writing all of the document content. If you select the second method, you can use either method to write any nested segments. Again, if while writing nested segments, you select the first method, you must continue to use that method, and so on.

   a. Call the PUT AGGREGATE routine with a completed aggregate of type DDIF$_SEG, whose DDIF$_SEG_CONTENT item references a sequence of aggregates that make up the entire content for that segment, including any nested segments. Using this method, you need only call the PUT AGGREGATE routine once, because the DDIF$_SEG aggregate written in the call to PUT AGGREGATE is already completely populated.

   b. Call the ENTER SCOPE routine, specifying **scope-code** as DDIF$K_SEGMENT_SCOPE, with a completed aggregate of type DDIF$_SEG whose DDIF$_SEG_CONTENT item is empty. You can then call the PUT AGGREGATE routine for each aggregate that makes up the segment content, in order. Once that segment and all its nested segments have been output, call the LEAVE

SCOPE routine, specifying **scope-code** as DDIF$K_SEGMENT_SCOPE, to complete that segment.

6 Call the LEAVE SCOPE routine, specifying **scope-code** as DDIF$K_CONTENT_SCOPE.

7 Call the LEAVE SCOPE routine, specifying **scope-code** as DDIF$K_DOCUMENT_SCOPE.

4.8 CDA Converters

The OPEN CONVERTER routine activates a front end module that processes files of a specified format. This format can be the same or a different format from that of the file currently being processed. Any processing options that were specified to the CONVERT routine for the document format are retrieved and appended to the item list that is specified for this routine.

5 Writing Converter Front and Back Ends

As described in Chapter 2, the CDA Converter actually comprises the converter kernel and a collection of front and back ends that process the supported input and output file-encoding formats. This chapter describes the techniques involved in writing converter front ends and back ends.

5.1 Document Conversion

Document conversion is accomplished by using the DIGITAL Document Interchange Format. The CDA Converter reads the input file and translates it to a CDA in-memory format, which is then translated to the specified output format. This conversion can be accomplished using either of two methods:

- In *document method conversion*, the entire document is read into memory before being converted.
- In *aggregate (incremental) method conversion*, each aggregate (along with its subaggregates) is converted individually so that, at any given time, only a few aggregates are available in memory.

Regardless of which method you use to perform document conversion, front ends and back ends convert a document of the specified input format to a document of the specified output format, using DDIF as the intermediate (in-memory) format. In general, front and back ends should use the incremental conversion method when the characteristics of the output document format are such that the entire input document does not have to be available in memory in order to be converted.

The basic steps involved in document conversion are as follows:

1. The user invokes the converter by issuing either the CONVERT/DOCUMENT DCL command or a call to the CONVERT routine. Regardless of which interface is used, the initial steps in the conversion process are performed by the converter.
2. The converter kernel invokes the main entry point in the front end (DDIF$READ_*format*) so that the front end is initialized and an input file or stream is opened.
3. The converter kernel then calls the main entry point in the appropriate back end (DDIF$WRITE_*format*) so that an output stream (and possibly an output file) is created.
4. The back end calls the converter kernel and requests information (either the entire document or an aggregate from the document) from the front end.

a. In the aggregate (incremental) conversion method, the following steps must be performed to actually translate the input document to the specified output format:

- The back end calls the CONVERT AGGREGATE routine to request an aggregate.
- The converter kernel calls the front end *get-aggregate* entry point in the front end to retrieve the requested aggregate.
- The front end *get-aggregate* routine reads enough information from the input stream to create a valid DDIF aggregate. This aggregate is then returned to the converter kernel.
- The converter kernel passes control, and the requested aggregate, to the back end.
- The back end translates the aggregate data to the specified output format and writes the information to the output stream.

The back end repeats these steps until the converter kernel returns an end-of-document status. The back end then closes the output stream, performs cleanup operations, and passes control to the converter kernel.

b. In the document conversion method, a single call to the CONVERT DOCUMENT routine performs all of the steps outlined in *a*. On return from a call to this routine, the entire document is available in memory. The back end translates the document data to the specified output format and writes the information to the output stream.

5 The converter kernel calls the *close* entry point in the front end. The front end then closes the input stream, performs cleanup operations, and returns control to the converter kernel.

6 The converter kernel performs final cleanup operations and returns control to its caller (the command line interface).

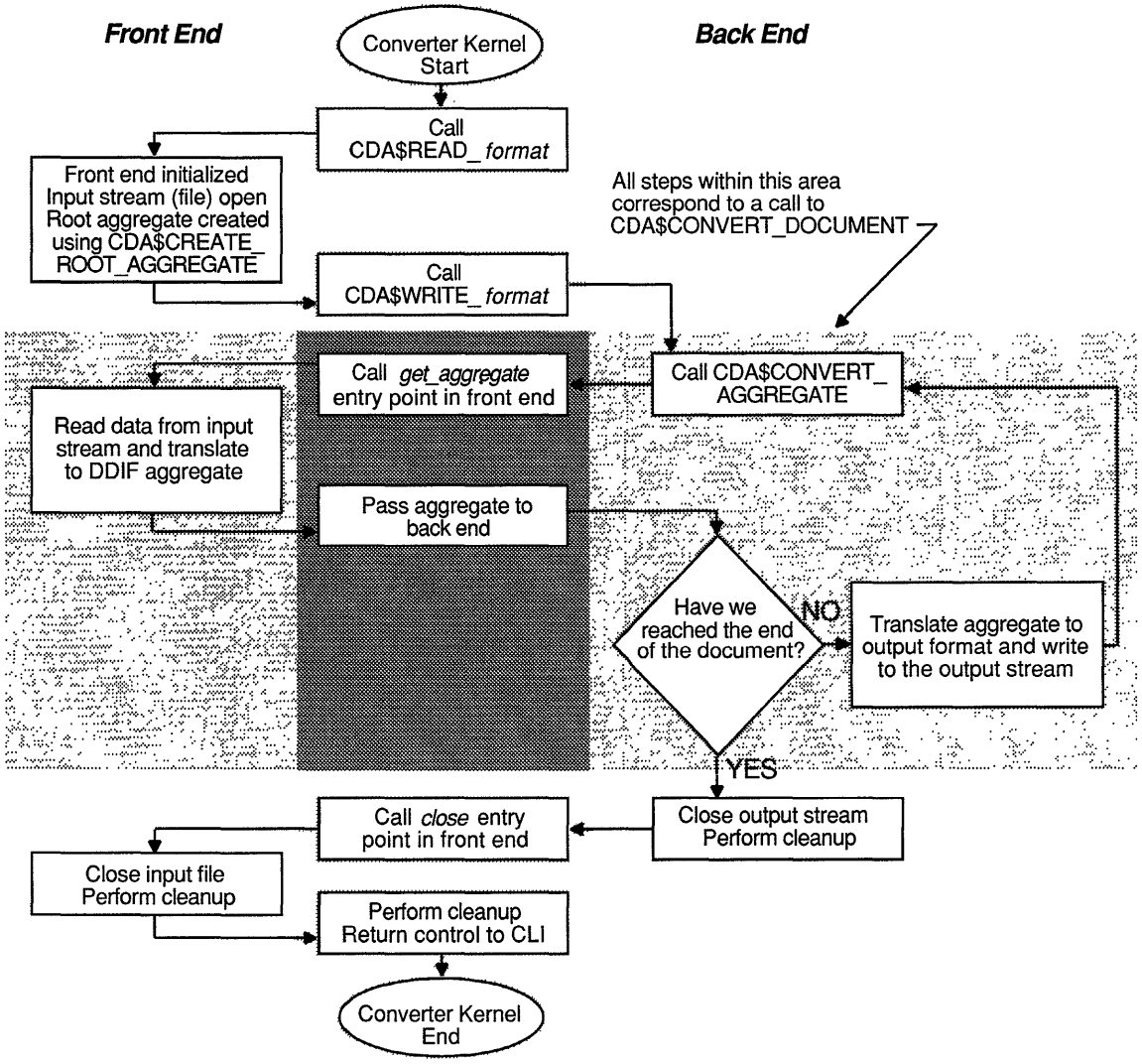
Figure 5–1 illustrates the basic flowchart of document conversion. The following sections discuss some programming guidelines that you should follow when writing front and back ends.

5.2 Front End

Each front end must meet certain criteria in order to work properly with the CDA Converter Kernel and with the supported back ends. Some of the recommendations that should be followed in order to ensure cooperation between the front end, kernel, and back end are as follows:

- To minimize memory usage, you should use the aggregate conversion method unless the desired document output format is such that the entire document must be in memory in order for the conversion to be performed.

Figure 5–1 Document Conversion Flowchart



ZK–0285A–GE

- You should ensure that the front end is reentrant so that multiple front ends can be invoked at one time.
- You should use the C programming language to develop the front and back ends, thus providing ease of portability between operating systems.

In addition, you must follow the predefined formats for the front end entry points as outlined in the following sections.

5.2.1 DDIF$READ_*format* Entry Point

The DDIF$READ_*format* entry point is the initial entry point in the front end. This routine initializes the conversion process and establishes any special processing information for the front end. The term *format* in the entry point name refers to the name of the document format that is read by this particular front end. For example, the entry point for the Text front end is DDIF$READ_TEXT.

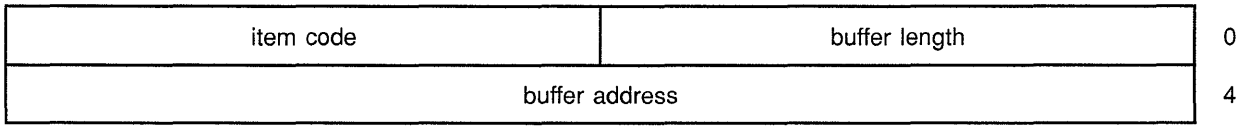
The main entry point for a front end must have the following format:

DDIF$READ_*format* standard-item-list
,converter-context ,front-end-context
,get-aggregate-procedure
,get-position-procedure
,close-procedure

The arguments are defined as follows:

standard-item-list
VMS usage: **item_list_2**
type: **record**
access: **read only**
mechanism: **by reference, array reference**
An item list that identifies the document source and may also contain options to control processing. The **standard-item-list** argument is the address of this item list.

Each entry in the item list is a 2-longword structure with the following format:



To terminate the item list, you must specify the final entry or longword as zero. Valid code values for the items in the front end **standard-item-list** are as follows:

- CDA$_INPUT_FILE

  The address and length of the file specification of the input document.

- CDA$_INPUT_DEFAULT

  The address and length of a string that specifies the default input file type. To simplify the porting of applications, the string should consist of only a file type in lowercase characters. If this parameter is omitted, the front end must supply an appropriate default file specification.

- CDA$_INPUT_PROCEDURE

  The address of a user *get* procedure that provides input. The item list length field must be set to 0. The input procedure must conform to the requirements for a user *get* routine. The calling sequence for a user *get* routine is defined in Section 5.3.

- CDA$_INPUT_PROCEDURE_PARM

  The address of a longword parameter to the input procedure. The item list length field must be set to 4.

- CDA$_INPUT_POSITION_PROCEDURE

  The parameter is the address of a procedure that provides position information. The item-list length field must be set to 0. The *get-position* procedure is specified in Section 5.2.3.

- CDA$_PROCESSING_OPTION

  The address and length of a string that contains an option to control processing. The format name and leading spaces and tabs have been removed from the string. This item code may occur more than once in the item list.

Either the CDA$_INPUT_FILE item or the CDA$_INPUT_PROCEDURE item, but not both, must occur once in the item list. If the CDA$_INPUT_PROCEDURE item is specified, then a single value for CDA$_INPUT_PROCEDURE_PARM can also be specified.

converter-context
VMS usage: **context**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**
Converter context required to call the OPEN CONVERTER routine. The **converter-context** argument is the address of an unsigned longword that contains this context.

front-end-context
VMS usage: **context**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives a front-end-defined value that identifies this particular instance of the front end. The **front-end-context** argument is the address of an unsigned longword that receives this context. This value is returned to the **get-aggregate-procedure** and the **close-procedure** arguments described later. All writable memory used by the front end must be allocated from dynamic memory and located by reference to this value.

get-aggregate-procedure
VMS usage: **procedure**
type: **procedure entry mask**
access: **write only**
mechanism: **by reference**
Receives the address of the *get-aggregate* routine. The **get-aggregate-procedure** argument receives the address of this procedure entry mask. The calling sequence for the *get-aggregate* routine is described in Section 5.2.2.

get-position-procedure
VMS usage: **procedure**
type: **procedure entry mask**
access: **write only**
mechanism: **by reference**
Receives the address of the *get-position* routine. The **get-position-procedure** argument receives the address of this procedure entry mask. The calling sequence for the *get-position* routine is described in Section 5.2.3.

close-procedure
VMS usage: **procedure**
type: **procedure entry mask**
access: **write only**
mechanism: **by reference**
Receives the address of the *close* routine. The **close-procedure** argument receives the address of this procedure entry mask. The calling sequence for the *close* routine is described in Section 5.2.4.

The possible status codes that DDIF$READ_*format* can return are either CDA$_NORMAL or any front end-specific error conditions.

5.2.2 *Get-Aggregate* Entry Point

The *get-aggregate* routine returns the next aggregate in the document to the converter kernel. Depending on the conversion method used, the *get-aggregate* routine either creates and populates the next document content aggregate (see Section 5.2.6.1) or it reads the next aggregate from the in-memory document (see Section 5.2.5.2). In either case, the returned aggregate must not be part of a sequence, and the DDIF$_SEG_CONTENT item of a DDIF$_SEG aggregate must be empty; the content must be returned one aggregate at a time followed by a DDIF$_EOS (end of segment) aggregate.

The call format for the *get-aggregate* routine is as follows:

get-aggregate-procedure front-end-context ,aggregate-handle ,aggregate-type

The arguments for this entry point are defined as follows:

front-end-context
VMS usage: **context**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**
Context returned from DDIF$READ_*format*. The **front-end-context** argument is the address of an unsigned longword that contains this context. Typically, this argument is used to specify the type of content aggregate to be created by the *get-aggregate* routine.

aggregate-handle
VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives the handle of the created and populated aggregate. The **aggregate-handle** argument is the address of an unsigned longword that receives this aggregate handle. This handle must be used in all subsequent operations on that aggregate.

aggregate-type
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives the aggregate type. The **aggregate-type** argument is the address of an unsigned longword that receives this aggregate type. If the aggregate is of type DDIF$_EOS (end of segment), **aggregate-handle** is 0.

The possible status codes that the *get-aggregate* routine can return are as follows:

CDA$_NORMAL	Normal successful completion.
CDA$_ENDOFDOC	End of document.

The *get-aggregate* routine can also return any front end-specific error conditions. Note that the *get-aggregate* routine must return the status CDA$_ENDOFDOC when the document has been completely transferred.

5.2.3 *Get-Position* Entry Point

The *get-position* routine returns the current position in and total size of the current data stream. The call format for this routine is as follows:

get-position-procedure front-end-handle ,stream-position
,stream-size

The arguments for this entry point are defined as follows:

front-end-handle
VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**
Identifier of the front end. The **front-end-handle** argument is the address of an unsigned longword that contains this handle. The front end handle is returned by DDIF$READ_*format*.

stream-position
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives the current position (in bytes) as measured from the start of the input stream being processed. The **stream-position** argument is the address of an unsigned longword that receives this position.

stream-size
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives the total size (in bytes) of the input stream being processed. The **stream-size** argument is the address of an unsigned longword that receives this size.

5.2.4 *Close* Entry Point

The *close* routine is used to terminate the operation of a front end by closing all open files and releasing all dynamic memory and other resources that have been allocated by the front end.

The call format for a *close* procedure is as follows:

close-procedure front-end-context

The argument for this entry point is defined as follows:

front-end-context
VMS usage: **context**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**
Context returned by DDIF$READ_*format*. The **front-end-context** argument is the address of an unsigned longword that contains this context. This context must specify the input file or stream to be closed.

The possible status codes that the *close* routine can return are either CDA$_NORMAL or any input converter-specific error conditions.

5.2.5 Front End Document-Method Conversion

When a front end performs document conversion by reading the entire document into memory, it typically follows these steps:

1 After being invoked by the CDA Converter Kernel, the front end performs the following initialization steps:

   a. Allocates a context block that stores pertinent information in dynamic memory. The context block typically stores information such as file, stream, and root aggregate handles, status, buffers, and other information extracted from the processed item list.

b. Processes the item list supplied by the **standard-item-list** argument to the CONVERT routine.

c. Processes any processing options specified.

d. Opens an input file.

e. Creates the document root aggregate (by calling the CREATE ROOT AGGREGATE routine).

f. Reads data from the input stream and creates the entire document in memory.

g. Closes the input stream (and, if applicable, the input file).

At this point, the entire document is in memory.

2 When the converter kernel invokes the *get-aggregate* entry point in the front end, the front end uses the PRUNE AGGREGATE routine to read each aggregate, pass it to the converter kernel (and thus to the back end), and remove it from the in-memory document after it has been processed. This step is repeated until the entire document is converted.

3 Upon completion of document conversion, the front end deletes the root aggregate from the in-memory document and deallocates the context block, and then returns control to the converter kernel.

The following sections discuss in more detail the steps that should be performed in each entry point of the front end.

5.2.5.1 DDIF$READ_*format* Routine

In order to initialize a document-method conversion, the DDIF$READ_*format* routine must first process the user-supplied item list, storing all pertinent information in the context block. The item list structure that is used to pass this information between the front end, back end, and the kernel is created by the CDA Converter Kernel; this structure contains the following fields:

- CDA$W_ITEM_LENGTH specifies the length of the item.
- CDA$W_ITEM_CODE specifies the item code, selected from the list specified in Section 5.2.1.
- CDA$W_ITEM_ADDRESS specifies the address of the item.

These fields are defined in the file CDA$DEF.SDL.

In addition, the DDIF$READ_*format* routine must process any specified processing options that the user selects for this conversion. If the format of the input file is not DDIF or Text, the front end must supply its own file-opening capability, typically through the use of the RMS $OPEN service, or the **open** C run-time library routine or equivalent language statement.

It is also recommended that the DDIF$READ_*format* routine define values for at least the following aggregate items:

- DDIF$_DSC_PRODUCT_IDENTIFIER specifies the registered facility mnemonic for the product that encoded the document.

- DDIF$_DSC_PRODUCT_NAME specifies the name of the product that encoded the document.

The DDIF$READ_*format* routine must call the CREATE ROOT AGGREGATE routine to create the document root aggregate. In the case of document-method conversion, the DDIF$READ_*format* routine must also create the DDIF$_DSC, DDIF$_DHD, and DDIF$_SEG aggregates before reading the entire document from the input stream and placing it in memory. Once the entire document is in memory, the DDIF$READ_*format* routine must close the input stream (and, if applicable, the input file). Again, if the format of the input file is not DDIF or Text, the DDIF$READ_*format* routine must supply its own file-closing capability, typically through the use of the RMS $CLOSE service, or the **close** C run-time library routine or equivalent language statement. At this point, control passes back to the converter kernel.

5.2.5.2 *Get-Aggregate* Routine

A front end should create aggregates on demand, rather than first creating the entire document in memory. However, if the entire document must be available in memory in order for the conversion to take place, the *get-aggregate* routine must use the PRUNE AGGREGATE routine to return the next content aggregate from the in-memory document. The PRUNE AGGREGATE routine removes the next sequential document content aggregate from an existing in-memory DDIF document and returns the aggregate identifier and type.

5.2.5.3 *Get-Position* Routine

The *get-position* routine provides a method for a back end to determine the total size of the current input stream, as well as to determine the current position within the stream. This routine is useful for viewer back ends that provide a scroll bar indicating the current position in the document being viewed.

5.2.5.4 *Close* Routine

In document-method conversion, the input file or stream has already been closed by the DDIF$READ_*format* routine. Therefore, the *close* routine simply performs regular cleanup operations and returns control to the CDA Converter Kernel.

5.2.6 Front End Aggregate-Method Conversion

When a front end performs document conversion by reading each aggregate into memory, it typically follows these steps:

1 After being invoked by the CDA Converter Kernel, the front end performs the following initialization steps:

   a. Allocates a context block that stores pertinent information in dynamic memory. The context block typically stores information such as file, stream, and root aggregate handles, status, buffers, and other information extracted from the processed item list.

   b. Processes the item list supplied by the **standard-item-list** argument to the CONVERT routine.

c. Processes any processing options specified.

d. Opens an input file.

e. Creates the document root aggregate (by calling the CREATE ROOT AGGREGATE routine).

2 When the kernel invokes the *get-aggregate* entry point in the front end, the front end reads enough information from the input file to complete a single content aggregate and its subaggregates. The front end then creates the appropriate aggregates, fills in the required information, and passes the completed aggregate back to the kernel. In this way, only a few aggregates at a time are in memory.

3 Once all of the information from the input file has been read and converted to the CDA in-memory format, the front end closes the input stream (and file, if appropriate), deallocates the context block, and returns control to the CDA Converter Kernel.

In order to initialize an aggregate-method conversion, the DDIF$READ_*format* routine must first process the user-supplied item list, storing all pertinent information in the context block. The item list structure that is used to pass this information between the front end, back end, and kernel is created by the CDA Converter Kernel; this structure contains the following fields:

- CDA$W_ITEM_LENGTH specifies the length of the item.
- CDA$W_ITEM_CODE specifies the item code, selected from the list specified in Section 5.2.1.
- CDA$W_ITEM_ADDRESS specifies the address of the item.

These fields are defined in the file CDA$DEF.SDL.

In addition, the DDIF$READ_*format* routine must process any specified processing options that the user selected for this conversion. If the format of the input file is not DDIF or Text, the front end must supply its own file-opening capability, typically through the use of the RMS $OPEN service, or the **open** C run-time library routine or equivalent language statement.

The DDIF$READ_*format* routine should also define values for at least the following aggregate items:

- DDIF$_DSC_PRODUCT_IDENTIFIER specifies the registered facility mnemonic for the product that encoded the document.
- DDIF$_DSC_PRODUCT_NAME specifies the name of the product that encoded the document.

The DDIF$READ_*format* routine must call the CREATE ROOT AGGREGATE routine to create the document root aggregate. Once the root aggregate is created, control passes back to the kernel.

5.2.6.1 *Get-Aggregate* Routine

Before creating any of the document content aggregates, the *get-aggregate* routine must first create a DDIF$_DSC aggregate, a DDIF$_DHD aggregate, and a DDIF$_SEG aggregate. Once these aggregates are created and the appropriate items have been stored (using the STORE ITEM routine), the *get-aggregate* routine creates and populates each sequential document content aggregate (and its subaggregates) that results from the translation of the input document. Once these aggregates are created and populated, the *get-aggregate* routine returns the handle and type of the parent aggregate. The aggregate type created must be a top-level content type, as listed in Table 5–1.

Table 5–1 Top-Level Aggregate Types

Aggregate Type	Meaning
DDIF$_DSC	Document descriptor
DDIF$_DHD	Document header
DDIF$_SEG	Document segment
DDIF$_TXT	Text content
DDIF$_GTX	General text content
DDIF$_HRD	Hard directive
DDIF$_SFT	Soft directive
DDIF$_HRV	Hard value directive
DDIF$_SFV	Soft value directive
DDIF$_BEZ	Bézier curve content
DDIF$_LIN	Polyline content
DDIF$_ARC	Arc content
DDIF$_FAS	Fill area set content
DDIF$_IMG	Image content
DDIF$_CRF	Content reference
DDIF$_EXT	External content
DDIF$_PVT	Private content
DDIF$_GLY	Layout galley
DDIF$_EOS	End of segment

5.2.6.2 *Get-Position* Routine

The *get-position* routine provides a method for a back end to determine the total size of the current input stream, as well as to determine the current position within the stream. This routine is useful for viewer back ends that provide a scroll bar indicating the current position in the document being viewed.

5.2.6.3 *Close* Routine

In aggregate-method conversion, the *close* routine must close the currently open file or stream in addition to performing the regular cleanup work. If the format of the input file is not DDIF or Text, the front end must supply its own file-closing capability, typically through the use of the RMS $CLOSE service, or the **close** C run-time library routine or equivalent language statement. Once all cleanup work has been completed, the *close* routine passes control back to the CDA Converter Kernel.

5.3 User-Supplied Input Procedures

The **get-rtn** and **get-prm** arguments are used to invoke a user stream *get* routine and to supply an argument to that routine. This routine reads bytes from an input stream. The application that creates the *get* routine also creates the buffer. Therefore, the application determines the buffer size and buffer management techniques. The caller of the *get* routine (namely, the CDA Toolkit) treats the buffer as read-only; it must contain valid data until the next call to the *get* routine.

The call format for a user *get* routine is as follows:

get-rtn get-prm ,num-bytes ,buf-adr

get-prm
VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**
User context argument. The **get-prm** argument contains the value of the parameter to be passed to the user *get* routine.

num-bytes
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives the number of bytes contained in the buffer. The **num-bytes** argument is the address of an unsigned longword that receives this number. The number of bytes is zero only if the stream does not contain any more data.

buf-adr
VMS usage: **address**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives the address of the buffer. The **buf-adr** argument is the address of an unsigned longword that receives the buffer address.

5.4 Back End Routine

Each back end must meet certain criteria in order to work properly with the converter kernel and with the supported back ends. Some of the recommendations that should be followed in order to ensure cooperation between the back end, CDA Converter Kernel, and front end are as follows:

- To minimize memory usage, you should use the aggregate conversion method unless the desired output format of your document is such that the entire document must be in memory in order to perform the conversion.
- You should use the C programming language to develop the front and back ends, thus providing ease of portability between operating systems.
- Unless your application is a document viewer, you should ensure that the conversion process runs to completion and never returns the CDA$_SUSPEND status. The standard VMS DCL command interface recalls the output converter procedure immediately in this situation.

In addition, you must follow the predefined format for the back end entry point as outlined in the following section.

5.4.1 DDIF$WRITE_*format* Entry Point

The DDIF$WRITE_*format* entry point is the entry point in the back end. This routine requests aggregates from the front end, converts them from the CDA in-memory format to the specified output format, and writes the information to the specified output file. The term *format* in the entry point name refers to the name of the document format that is being written by this particular back end. For example, the entry point for the Text back end is DDIF$WRITE_TEXT.

The call format for a DDIF$WRITE_*format* routine is as follows:

DDIF$WRITE_*format* function-code ,standard-item-list
,private-item-list ,front-end-handle
,back-end-context

function-code
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Symbolic constant that identifies the function to be performed. The **function-code** argument is the address of an unsigned longword that contains this symbolic constant. These constant values are defined in file CDA$DEF.SDL. Valid values are as follows:

- CDA$_START

  Start conversion. This function code must be specified to begin a document conversion.

- CDA$_CONTINUE

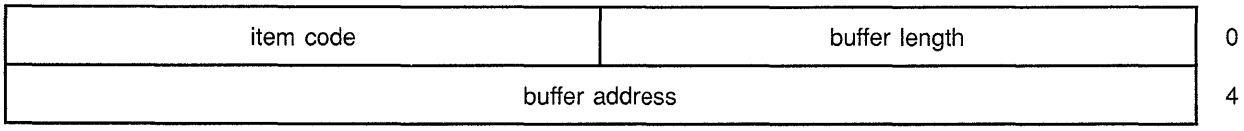
  Continue a conversion that was suspended. This function code may only be specified if a previous call to DDIF$WRITE_*format* returned the value CDA$_SUSPEND. If CDA$_SUSPEND is returned by a call to DDIF$WRITE_*format*, either CDA$_CONTINUE or CDA$_STOP must be specified so that resources locked by the conversion may be released.

- CDA$_STOP

  Discontinue a conversion that was suspended. This function code may only be specified if the previous call to DDIF$WRITE_*format* returned the value CDA$_SUSPEND. If CDA$_SUSPEND is returned by a call to DDIF$WRITE_*format*, either CDA$_STOP or CDA$_CONTINUE must be specified so that resources locked by the conversion may be released.

standard-item-list
VMS usage: **item_list_2**
type: **record**
access: **read only**
mechanism: **by reference, array reference**
An item list that identifies the document destination and may also contain options to control processing. The **standard-item-list** argument is the address of this item list.

Each entry in the item list is a 2-longword structure with the following format:



To terminate the item list you must specify the final entry or longword as zero. The **standard-item-list** argument is ignored when **function-code** is set to either CDA$_CONTINUE or CDA$_STOP. Valid code values for the items in the **standard-item-list** are as follows:

- CDA$_OUTPUT_FILE

  The address and length of the file specification of the output document.

- CDA$_OUTPUT_DEFAULT

  The address and length of the default file specification of the output document. If this parameter is omitted, the back end must supply an appropriate default file specification.

- CDA$_OUTPUT_PROCEDURE

  The address of a procedure to receive output. The item list length field must be set to 0. The output procedure must conform to the requirements for a user *put* routine. The calling sequence for a user *put* routine is defined in Section 5.4.2.

- CDA$_OUTPUT_PROCEDURE_PARM

  The address of a longword parameter to the output procedure. The item list length field must be set to 4.

- CDA$_OUTPUT_PROCEDURE_BUFFER

  The address and length of the initial output buffer for the output procedure.

- CDA$_PROCESSING_OPTION

  The address and length of a string that contains an option to control processing. The format name and leading spaces and tabs have been removed from the string. This item code may occur more than once in the item list.

Either CDA$_OUTPUT_FILE or CDA$_OUTPUT_PROCEDURE, but not both, must occur once in the item list. If the CDA$_OUTPUT_PROCEDURE item occurs, then the CDA$_OUTPUT_PROCEDURE_PARM item and the CDA$_OUTPUT_PROCEDURE_BUFFER item may each occur once in the item list.

private-item-list
VMS usage: **unspecified**
type: **unspecified**
access: **read only**
mechanism: **by reference**
A private item list that is passed directly to the back end. The **private-item-list** argument is the address of this private item list. The specification of this item list is the responsibility of the back end. Its purpose is to provide for direct two-way communication between the caller of the CONVERT routine and the back end.

front-end-handle
VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**
Identifier of the front end that will process the document content. The **front-end-handle** argument is the address of an unsigned longword that contains this front end handle. This handle is passed to either the CONVERT DOCUMENT routine or the CONVERT AGGREGATE routine.

back-end-context
VMS usage: **context**
type: **longword (unsigned)**
access: **read only or write only**
mechanism: **by reference**
When **function-code** is set to CDA$_START, this argument receives a value defined by the back end that identifies this particular instance of the back end. The **back-end-context** argument is the address of an unsigned longword that either receives or specifies the converter context. This value will be returned to DDIF$WRITE_*format* for the functions CDA$_CONTINUE and CDA$_STOP. If a back end returns CDA$_SUSPEND, all writable memory used by the back end must be allocated from dynamic memory and located by reference to this value.

The possible status codes that DDIF$WRITE_*format* can return are as follows:

CDA$_NORMAL	Normal successful completion.
CDA$_SUSPEND	Converter is suspended.
CDA$_INVFUNCOD	Invalid function code.
CDA$_INVITMLST	Invalid item list.
CDA$_UNSUPFMT	Unsupported document format.

DDIF$WRITE_*format* can also return any error returned by the specific front end or the specific back end.

In order for the back end to call through to the front end, two routines are provided:

- The CONVERT DOCUMENT routine invokes the document-method conversion of an input file to the specified output format.
- The CONVERT AGGREGATE routine invokes the aggregate-method conversion of an input file to the specified output format.

The back end must use one of these routines to request the appropriate information from the front end.

If the format of the output file is not DDIF or Text, the back end must supply its own file-creation capability, typically through the use of the **creat** C run-time library routine or equivalent language statement.

In order to initialize a document-method conversion, the DDIF$WRITE_*format* routine must first process the user-supplied item list, storing all pertinent information in the context block. The item list structure that is used to pass this information between the front end, back end, and kernel is created in the CDA Converter Kernel; this structure contains the following fields:

- CDA$W_ITEM_LENGTH specifies the length of the item.
- CDA$W_ITEM_CODE specifies the item code, selected from the list specified in this section.
- CDA$W_ITEM_ADDRESS specifies the address of the item.

These fields are defined in the file CDA$DEF.SDL.

5.4.2 User-Supplied Output Procedures

The **put-rtn** and **put-prm** arguments are used to invoke a user stream *put* routine, and to supply an argument to that routine. The call format for this user routine is as follows:

put-rtn put-prm ,num-bytes ,buf-adr ,next-buf-len ,next-buf-adr

The arguments for this routine are defined as follows:

put-prm
VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**
User context argument. The **put-prm** argument is the value of the parameter to be passed to the user *put* routine.

num-bytes
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**
Number of bytes contained in the buffer. The **num-bytes** argument is the address of an unsigned longword that contains this value.

buf-adr
VMS usage: **vector_byte_unsigned**
type: **byte (unsigned)**
access: **read only**
mechanism: **by reference, array reference**
Address of the buffer. The **buf-adr** argument is the address of an array of unsigned bytes.

next-buf-len
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Length of the buffer specified by **next-buf-adr**. The **next-buf-len** argument is the address of an unsigned longword that receives this length.

next-buf-adr
VMS usage: **address**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Address of a buffer that will receive further output data. The **next-buf-adr** argument is the address of an unsigned longword that receives this address. **Next-buf-adr** may simply be the current buffer, or a different buffer.

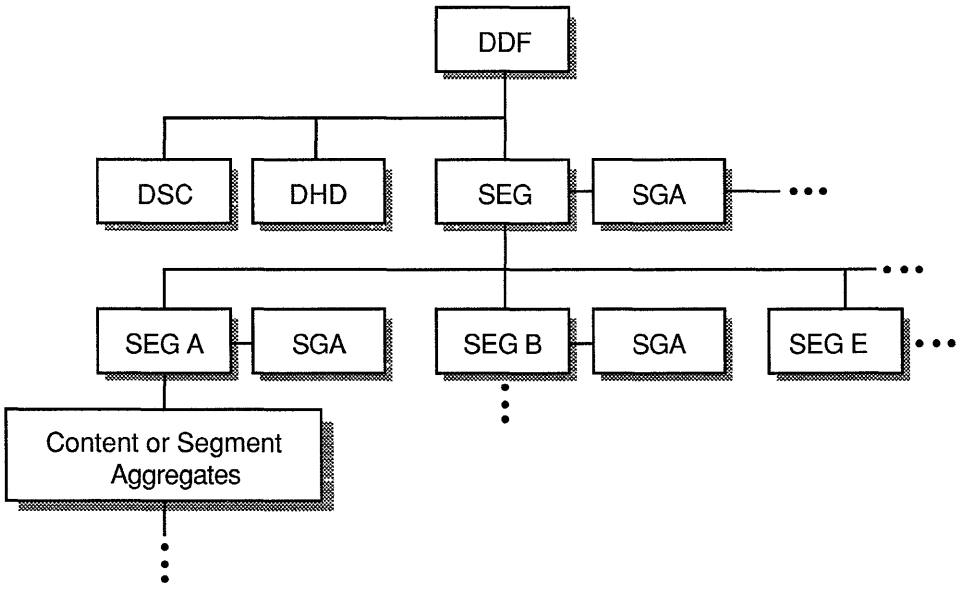
6 DDIF Structures

This chapter provides an overview of the general structure of a DDIF document, and then provides detailed references for each DDIF-supported aggregate structure.

6.1 DDIF Document Structure Overview

Every DDIF document has the same general structure. The document must have a root aggregate, a document descriptor aggregate, a document header aggregate, and content. It is the content that differentiates one document from another; however, the overall document structure is the same and is shown in Figure 6–1.

Figure 6–1 Compound Document Structure



Each DDIF aggregate type and its corresponding items is discussed in this chapter. Appendix D contains the tables that define each DDIF aggregate and the aggregate item encodings.

6.2 Generic Aggregate Items

In addition to the items defined by each individual aggregate, DDIF also supports three "generic" aggregate items that can be specified for every aggregate described in this chapter. These items are described in Table 6–1.

Table 6–1 Generic Aggregate Items

Item Name	Encoding	Meaning
DDIF$_USER_CONTEXT	Longword	Specifies user context
DDIF$_AGGREGATE_TYPE	Word	Specifies the type of the aggregate
DDIF$_ALL_MAX	Longword	Specifies the number of items in the aggregate

6.3 Document Root Aggregate

The DDIF document root aggregate (type DDIF$_DDF) identifies this particular instance of a DDIF document. This aggregate contains the following items:

- A document descriptor item (type DDIF$_DDF_DESCRIPTOR) that describes the document encoding. This item is encoded as the handle of a DDIF$_DSC aggregate. For more information on the DDIF$_DSC aggregate, see Section 6.4.
- A document header item (type DDIF$_DDF_HEADER) that contains parameters and processing instructions that apply to the document as a whole. This item is encoded as the handle of a DDIF$_DHD aggregate. For more information on the DDIF$_DHD aggregate, see Section 6.5.
- A document content item (type DDIF$_DDF_CONTENT) that specifies the content of the document. This item is encoded as the handle of a DDIF$_SEG aggregate. This DDIF$_SEG aggregate specifies the root or parent segment of the document. For more information on the DDIF$_SEG aggregate, see Section 6.6.

Table 6–2 lists the items in a document root aggregate and their corresponding encodings.

Table 6–2 Document Root Aggregate (DDIF$_DDF)

Item Name	Item Encoding
DDIF$_DDF_DESCRIPTOR	Handle of DDIF$_DSC aggregate
DDIF$_DDF_HEADER	Handle of DDIF$_DHD aggregate
DDIF$_DDF_CONTENT	Handle of DDIF$_SEG aggregate

6.4 Document Descriptor

The document descriptor aggregate (type DDIF$_DSC) specifies the version level of the DIGITAL Document Interchange Format used by this document, and identifies the software that created the document. This aggregate contains the following items:

- A major version indicator (type DDIF$_DSC_MAJOR_VERSION) that acts as the primary indicator of compatibility between the current version of DDIF and the version of DDIF used to encode the document. This item is encoded as an integer.

  The literal DDIF$K_MAJOR_VERSION is defined to represent the highest major version supported by the CDA Toolkit. Applications should use this literal for the major version indicator. On output, the CDA Toolkit ignores the current value of this item and instead supplies the current version.

- A minor version indicator (type DDIF$_DSC_MINOR_VERSION) that specifies the revision number of the current DDIF encoding. This item is encoded as an integer.

  The literal DDIF$K_MINOR_VERSION is defined to represent the highest minor version supported by the CDA Toolkit. Applications should use this literal for the minor version indicator. On output, the CDA Toolkit ignores the current value of this item and instead supplies the current version.

- A product identifier item (type DDIF$_DSC_PRODUCT_IDENTIFIER) that contains a registered facility mnemonic representing the software that encoded the document. This item is encoded as a string.

  The product identifier can be an acronym or abbreviation for the product name. This identifier is constant across versions of the product. If a product places private segment tags in the document, the product identifier string is used to prefix those segment tags.

- A product name item (type DDIF$_DSC_PRODUCT_NAME) that indicates the name of the product that encoded the document. This item is encoded as an array of type character string so that, if desired, the product name can be specified in multiple languages.

  The product name string contains the version number of the product. The name of the product should be spelled in full, and should include a baselevel of version number.

Table 6–3 lists the items in a document descriptor aggregate and their corresponding encodings.

Table 6–3 Document Descriptor Aggregate (DDIF$_DSC)

Item Name	Item Encoding
DDIF$_DSC_MAJOR_VERSION	Integer
DDIF$_DSC_MINOR_VERSION	Integer
DDIF$_DSC_PRODUCT_IDENTIFIER	String
DDIF$_DSC_PRODUCT_NAME	Array of type character string

6.5 Document Header

The document header aggregate contains data that pertains to the document as a whole; it describes the document to processors that receive it. The DDIF document header aggregate (type DDIF$_DHD) contains the following items:

- An optional private header data item (type DDIF$_DHD_PRIVATE_DATA) that contains global information about the document not currently standardized by DDIF. This item is encoded as a sequence of DDIF$_PVT aggregates. (For more information on the DDIF$_PVT aggregate, see Section 6.15.2.) All interpretations of the private data are subject only to private agreements between the parties concerned.
- An optional title item (type DDIF$_DHD_TITLE) that contains the user-visible name of the document. This item is encoded as an array of type character string.
- An optional author item (type DDIF$_DHD_AUTHOR) that contains the name of the person or persons responsible for the information content of the document. This item is encoded as an array of type character string.
- An optional version item (type DDIF$_DHD_VERSION) that contains a character string used to distinguish this version of the document from all other versions. This item is encoded as an array of type character string.
- An optional date item (type DDIF$_DHD_DATE) that contains the date associated with this version of the document. This item is encoded as a string.
- An optional conformance tags item (type DDIF$_DHD_CONFORMANCE_TAGS) that contains a set of tags indicating the processing restrictions that apply to the document, and what subset of DDIF syntax has been used to describe the document. This item is encoded as an array of type string with *add-info*, where *add-info* can take the following values:

DDIF$K_PRIVATE_CONFORMANCE	Indicates nonstandard processing restrictions
DDIF$K_SRQ_CONFORMANCE	Indicates that the structure descriptions in this document were strictly observed

- An optional external references item (type DDIF$_DHD_EXTERNAL_REFERENCES) that contains a list of file names (or other system-specific file specifiers) that are referenced from within the document. This item is encoded as a sequence of DDIF$_ERF aggregates. (For more information on the DDIF$_ERF aggregate, see Section 6.17.) In the body of the document, external references are specified as indexes into this list.
- An optional languages indicator (type DDIF$_DHD_LANGUAGES_C) that specifies the natural languages and programming languages that are delineated for processing by language tools. This item is encoded as an array of type enumeration. Valid values are as follows:

DDIF$K_ISO_639_LANGUAGE	A string that selects a language and dialect that are specified using the ISO 639 Standard. In this case, the DDIF$_DHD_LANGUAGES item is encoded as a string.
DDIF$K_OTHER_LANGUAGE	A character string that indicates the language and dialect using a "user-readable" name; this is used for those languages and dialects not covered by the ISO 639 Standard. In this case, the DDIF$_DHD_LANGUAGES item is encoded as a character string.

- The optional language item (type DDIF$_DHD_LANGUAGES) that contains a list of the actual languages from the selected language type that are delineated for processing. This item is encoded as an array of type variable.

  If you specify DDIF$_DHD_LANGUAGES_C as DDIF$K_ISO_639_LANGUAGE, you must specify DDIF$_DHD_LANGUAGES as one of the natural languages defined by the ISO 639 Standard, specifying the language symbol and country code. The following table illustrates some common examples:

Language/Country	String
English/US	E/USA/
English/Britain	E/GB/
French/France	F/F/
German/Germany	D/D/

- An optional style guide item (type DDIF$_DHD_STYLE_GUIDE) that provides a reference to an external style guide that contains all or some of the presentation and layout attributes for the elements in the document. This item is encoded as an integer; it acts as an index into the DDIF$_DHD_EXTERNAL_REFERENCES item. The style guide may or may not be encoded in DDIF format.

Table 6–4 lists all of the items in a document header aggregate and their corresponding encodings.

Table 6–4 Document Header Aggregate (DDIF$_DHD)

Item Name	Item Encoding
DDIF$_DHD_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates
DDIF$_DHD_TITLE	Array of type character string
DDIF$_DHD_AUTHOR	Array of type character string
DDIF$_DHD_VERSION	Array of type character string
DDIF$_DHD_DATE	String
DDIF$_DHD_CONFORMANCE_TAGS	Array of type string with *add-info*
DDIF$_DHD_EXTERNAL_REFERENCES	Sequence of DDIF$_ERF aggregates
DDIF$_DHD_LANGUAGES_C	Array of type enumeration
DDIF$_DHD_LANGUAGES	Array of type variable
DDIF$_DHD_STYLE_GUIDE	Integer

6.6 Document Content

The content of a document is contained in a single segment called the **root segment**. The root segment, in turn, contains zero or more content segments or elements, including (but not restricted to) text, graphics, images, and nested segments. Each individual content segment aggregate type is discussed in a separate section of this chapter.

The standard content aggregates specify the basic contents of a document, including characters, lines, and pixels. Each of these content types can be presented on a video display or hardcopy device. The presentation style for the document content is governed by the presentation attributes specified for the segment in which the various aggregates are contained. By grouping the various aggregates in segments, you can create larger units (for example, paragraphs of text).

The document segment aggregate contains the following items:

- An optional segment identifier (type DDIF$_SEG_ID) that identifies the segment for reference from other segments. This item is encoded as a string.

  References to labeled segments are not limited to those segments nested under the labeled segment; labeled segments can be referenced from any segment in the document and from other documents. Note that segments should be labeled only if they are referenced.

- An optional segment user label item (type DDIF$_SEG_USER_LABEL) that specifies the user-assigned name of the particular segment of content. This item is encoded as an array of type character string. This string is only for use by the user; it cannot be used to reference the segment from other segments. A typical use of a user label would be to allow users to name graphic objects and manipulate them by name.

- An optional segment type item (type DDIF$_SEG_SEGMENT_TYPE) that references a segment type definition in the segment attributes of a parent segment or in the style guide. This item is encoded as a string.

  This string is equivalent to the string specified by the DDIF$_TYD_LABEL item in the type definition (DDIF$_TYD) aggregate. For more information on the DDIF$_TYD aggregate, see Section 6.27. Note that when a segment references a segment type, it acquires the attributes bound to the segment type.

- An optional segment attribute item (type DDIF$_SEG_SPECIFIC_ATTRIBUTES) that binds presentation and processing attributes to the segment, and defines generic types and content for reference from nested segments. This item is encoded as the handle of a DDIF$_SGA aggregate. For more information on the DDIF$_SGA aggregate, see Section 6.20.

- An optional segment generic layout item (type DDIF$_SEG_GENERIC_LAYOUT) that specifies an element of generic layout for the segment. This item is encoded as the handle of a DDIF$_LG1 aggregate. (For more information on the DDIF$_LG1 aggregate, see Section 6.33.) Note that this item can only be specified on the root segment of a document. Generic layout descriptions placed on segments other than the root segment are ignored.

- An optional segment specific layout item (type DDIF$_SEG_SPECIFIC_LAYOUT) that specifies an element of specific layout for the segment. This item is encoded as the handle of a DDIF$_LS1 aggregate. (For more information on the DDIF$_LS1 aggregate, see Section 6.34.) Note that this item can only be specified on the root segment of a document. Specific layout descriptions placed on segments other than the root segment are ignored.

- An optional segment content item (type DDIF$_SEG_CONTENT) that specifies the content of the segment. This item is encoded as a sequence of content. A sequence of content is a linked list of any of the following aggregate types:

DDIF$_ARC	DDIF$_BEZ	DDIF$_CRF
DDIF$_EXT	DDIF$_FAS	DDIF$_GTX
DDIF$_HRD	DDIF$_HRV	DDIF$_IMG
DDIF$_LIN	DDIF$_PVT	DDIF$_SEG
DDIF$_SFT	DDIF$_SFV	DDIF$_TXT

  The DDIF$_SEG_CONTENT item contains the handle of the first aggregate in the sequence of content aggregates.

Table 6–5 lists the items in a document segment aggregate and their corresponding encodings.

Table 6–5 Document Segment Aggregate (DDIF$_SEG)

Item Name	Item Encoding
DDIF$_SEG_ID	String
DDIF$_SEG_USER_LABEL	Array of type character string
DDIF$_SEG_SEGMENT_TYPE	String
DDIF$_SEG_SPECIFIC_ATTRIBUTES	Handle of DDIF$_SGA aggregate
DDIF$_SEG_GENERIC_LAYOUT	Handle of DDIF$_LG1 aggregate
DDIF$_SEG_SPECIFIC_LAYOUT	Handle of DDIF$_LS1 aggregate
DDIF$_SEG_CONTENT	Sequence of content

6.6.1 Content Categories

DDIF content is divided into categories. The content category of a segment is denoted by a tag on that segment or on a parent segment. The standard content category tags are as follows:

Content Category	Tag
Image	$I
Graphics	$2D
Text	$T
Table	$TBL
Page Description Language	$PDL

These content category tags are stored in the DDIF$_SGA_CONTENT_CATEGORY item of the segment attributes aggregate (type DDIF$_SGA). For more information on the segment attributes aggregate, see Section 6.20. Other standard content category tags are reserved.

The content category constrains the content types that can occur within the categorized segment. For example, the graphics content category ($2D) is restricted to the graphics and text content. To include an image in the context of the graphics segment, the image must be contained in a segment tagged with the image content category ($I).

Content is always represented in a standard encoding without respect to content category. For example, graphics text is represented in the same encoding as document text, although it may (but does not have to) be processed by a graphics processor rather than a text formatter.

Most content categories can be nested. Graphics can be nested within document text, and document text within graphics. Individual content categories can place restrictions on the way other categories are nested within them.

You can determine the content category of a segment through inheritance, through a generic type reference, or by actually binding a content category tag to the segment itself.

6.6.2 Segment Tags

Segment tags are used to indicate the processing characteristics of content, including relationships to user interfaces and indications of special constraints on content.

Two types of tags are allowed:

- **Standard tags** defined by DDIF
- **Private tags** defined by individual DDIF processors

A given segment can have both a standard tag and a private tag, in cases where the processing of the segment meets the basic criteria of the standard tag, but the private tag provides a finer granularity of the naming scheme or the potential for additional processing. For example, DDIF provides a paragraph tag ($P). An editor that allows two types of paragraphs might define two segment types, each with a $P tag, but with different private tags and different presentation attributes.

Tags have no effect on the presentation of content. Instead, they can denote public or private rules about editing behavior, whether content is included in indexes, and other abstract relationships between content elements in the segment.

6.6.3 Presentation Attributes of Content

The rendering of document content is specified by the presentation attributes bound to the segment, either directly or through inheritance mechanisms. It is not necessary for all attributes of content to be specified in the document; an initial state of attributes is defined by DDIF. Conceptually, each DDIF document is enclosed in a segment that establishes the default attributes for document content. Creators can specify the defaults for the content by binding them to the root segment.

6.7 Text Content

Textual document content consists of graphics characters and spaces from standard and private character sets. Format directives such as **new-line** and **new-page** are expressed as DDIF-defined directives.

Text presentation style is controlled by **text attributes**. For more information on text attributes, see Section 6.20.15. Text layout is specified by **layout attributes**. For more information on layout attributes, see Section 6.36.

Text that is wrapped, formatted, and paginated is limited to the $T content category. Text that is imaged along a path is limited to the $2D content category. Text content and presentation attributes are specified identically in both document text categories; only the layout differs.

6.7.1 Latin1 Text Content

The Latin1 text content aggregate (type DDIF$_TXT) contains the text content item (type DDIF$_TXT_CONTENT) that indicates that the character set to be used is Latin1. This item is encoded as a string.

Table 6–6 lists the item in a Latin1 text content aggregate and its corresponding encoding.

Table 6–6 Latin1 Text Content Aggregate (DDIF$_TXT)

Item Name	Item Encoding
DDIF$_TXT_CONTENT	String

6.7.2 General Text Content

The general text content aggregate (type DDIF$_GTX) contains the text content item (type DDIF$_GTX_CONTENT) that indicates the character set to be used. This item is encoded as a character string.

Table 6–7 lists the item in a general text content aggregate and its corresponding encoding.

Table 6–7 General Text Content Aggregate (DDIF$_GTX)

Item Name	Item Encoding
DDIF$_GTX_CONTENT	Character string

The valid values for the character set identifier are listed in Table 6–8.

Table 6–8 Character Set Identifiers

Identifier	Character Set
DDIF$K_ISO_LATIN1	ISO Latin 1
DDIF$K_ISO_LATIN2	ISO Latin 2
DDIF$K_ISO_LATIN6	ISO Latin 6
DDIF$K_ISO_LATIN7	ISO Latin 7
DDIF$K_ISO_LATIN8	ISO Latin 8
DDIF$K_JIS_KATAKANA	JIS Roman, JIS Katakana
DDIF$K_DEC_TECH	DEC Special Graphics, DEC Technical
DDIF$K_DEC_MATH_ITALIC	DEC Mathematics Italic
DDIF$K_DEC_MATH_SYMBOL	DEC Mathematics Symbol
DDIF$K_DEC_MATH_EXTENSION	DEC Mathematics Extension
DDIF$K_DEC_PUBLISHING	DEC Publishing

(continued on next page)

Table 6–8 (Cont.) Character Set Identifiers

Identifier	Character Set
DDIF$K_DEC_KANJI	DEC Kanji
DDIF$K_DEC_HANZI	DEC Hanzi

6.8 Directives

Directives guide the formatting of text. DDIF directives are either hard or soft, depending on whether they are requested by the user or inserted by software. All directives are restricted to the $T content category.

6.8.1 Hard Directive

The hard directive aggregate (type DDIF$_HRD) contains the hard directive item (type DDIF$_HRD_DIRECTIVE) that specifies a hard directive (for example, a user-specified page break). This item is encoded as an enumeration. The valid values for this enumeration are listed in Section 6.8.3.

Table 6–9 lists the item in a hard directive aggregate and its corresponding encoding.

Table 6–9 Hard Directive Aggregate (DDIF$_HRD)

Item Name	Item Encoding
DDIF$_HRD_DIRECTIVE	Enumeration

6.8.2 Soft Directive

The soft directive aggregate (type DDIF$_SFT) contains the soft directive item (type DDIF$_SFT_DIRECTIVE) that specifies a soft directive (for example, a software-inserted page break). This item is encoded as an enumeration. The valid values for this enumeration are listed in Section 6.8.3.

Table 6–10 lists the item in a soft directive aggregate and its corresponding encoding.

Table 6–10 Soft Directive Aggregate (DDIF$_SFT)

Item Name	Item Encoding
DDIF$_SFT_DIRECTIVE	Enumeration

6.8.3 Directive Values

The items DDIF$_HRD_DIRECTIVE, DDIF$_SFT_DIRECTIVE, and DDIF$_LL1_INITIAL_DIRECTIVE (from the layout attributes aggregate described in Section 6.36) can all take any one of the values listed in Table 6–11.

Table 6–11 Directive Values

Directive	Meaning
DDIF$K_DIR_NEW_PAGE	Begins a new page.
DDIF$K_DIR_NEW_LINE	Begins a new line of text.
DDIF$K_DIR_NEW_GALLEY	Begins a new layout galley (such as a column). Software that does not support galley layout interprets the new galley directive as a new page.
DDIF$K_DIR_TAB	Moves the horizontal text position to the next tab stop.
DDIF$K_DIR_SPACE	Is treated as a space in the current font. The space directive is usually soft, and is used to indicate that software has inserted a space between wrapped lines.
DDIF$K_DIR_HYPHEN_NEW_LINE	Specifies that the line break is preceded by a hyphen. This directive is typically soft, and is used to indicate that software has inserted a hyphen at the place it broke the line.
DDIF$K_DIR_WORD_BREAK_POINT	Identifies an embedded point at which a word may be broken, if need be, for justification.
DDIF$K_DIR_LEADERS	Inserts leader characters according to the current leader attributes. A leader directive is treated like a space during justification, except that leader characters are inserted instead of space. The rendering of leaders is controlled by the current leader attributes and other text attributes.
DDIF$K_DIR_BACKSPACE	Specifies that the first character following this directive should be centered over the last character imaged.
DDIF$K_NULL	Suppresses the inheritance of the initial-directive element of layout attributes. This directive has no effect on imaging or processing.
DDIF$K_DIR_NO_HYPHEN_WORD	Suppresses hyphenation until the next space character or space directive is encountered.

6.8.4 Hard Value Directive

The hard value directive aggregate (type DDIF$_HRV) is a hard directive that has a parametric value. The hard value directive aggregate contains the following items:

- A hard value directive indicator (type DDIF$_HRV_C) that specifies whether the hard value directive is an escapement directive or a variable reset directive. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_DIR_ESCAPEMENT	Indicates an escapement directive that specifies the relative or constant distance by which to increment the current text position. If you specify this value, you must supply values for the items DDIF$_HRV_ESC_RATIO_N through DDIF$_HRV_ESC_CONSTANT.
DDIF$K_DIF_VARIABLE_RESET	Indicates a variable reset directive that specifies a directive to reset the value of the specified variable. If you specify this value, you must supply values for the items DDIF$_HRV_RESET_VARIABLE through DDIF$_HRV_RESET_VALUE.

- An escapement ratio numerator item (type DDIF$_HRV_ESC_RATIO_N) that specifies the magnitude of a ratio that is multiplied by a context-dependent measurement to obtain a proportional measurement. This item is encoded as an integer.
- An escapement ratio denominator item (type DDIF$_HRV_ESC_RATIO_D) that specifies the units of precision used in the ratio. This item is encoded as an integer.
- An escapement constant indicator (type DDIF$_HRV_ESC_CONSTANT_C) that indicates whether the escapement constant is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An escapement constant item (type DDIF$_HRV_ESC_CONSTANT) that specifies the constant measurement to be used as an escapement. This item is encoded as a variable.
- A reset variable item (type DDIF$_HRV_RESET_VARIABLE) that specifies the label of the variable to be reset by the hard value directive. This item is encoded as a string.
- A reset value indicator (type DDIF$_HRV_RESET_VALUE_C) that indicates whether the hard value directive reset value is specified as a variable or constant value. This item is encoded as an expression enumeration.
- A reset value item (type DDIF$_HRV_RESET_VALUE) that specifies the new value of the variable. This item is encoded as a variable.

Table 6–12 lists the items in a hard value directive aggregate and their corresponding encodings.

Table 6–12 Hard Value Directive Aggregate (DDIF$_HRV)

Item Name	Item Encoding
DDIF$_HRV_C	Enumeration
DDIF$_HRV_ESC_RATIO_N	Integer
DDIF$_HRV_ESC_RATIO_D	Integer
DDIF$_HRV_ESC_CONSTANT_C	Measurement enumeration
DDIF$_HRV_ESC_CONSTANT	Variable
DDIF$_HRV_RESET_VARIABLE	String
DDIF$_HRV_RESET_VALUE_C	Expression enumeration
DDIF$_HRV_RESET_VALUE	Variable

6.8.5 Soft Value Directive

The soft value directive aggregate (type DDIF$_SFV) is a soft directive that has a parametric value. The soft value directive aggregate contains the following items:

- A soft value directive indicator (type DDIF$_SFV_C) that specifies whether the soft value directive is an escapement directive or a variable reset directive. This item is encoded as an enumeration. Valid values are as follows:

  DDIF$K_DIR_ESCAPEMENT — Indicates an escapement directive that specifies the relative or constant distance by which to increment the current text position. If you specify this value, you must supply values for the items DDIF$_SFV_ESC_RATIO_N through DDIF$_SFV_ESC_CONSTANT.

  DDIF$K_DIF_VARIABLE_RESET — Indicates a variable reset directive that specifies a directive to reset the value of the specified variable. If you specify this value, you must supply values for the items DDIF$_SFV_RESET_VARIABLE through DDIF$_SFV_RESET_VALUE.

- An escapement ratio numerator item (type DDIF$_SFV_ESC_RATIO_N) that specifies the magnitude of a ratio that is multiplied by a context-dependent measurement to obtain a proportional measurement. This item is encoded as an integer.
- An escapement ratio denominator item (type DDIF$_SFV_ESC_RATIO_D) that specifies the units of precision used in the ratio. This item is encoded as an integer.
- An escapement constant indicator (type DDIF$_SFV_ESC_CONSTANT_C) that indicates whether the escapement constant is specified as a variable or constant value. This item is encoded as a measurement enumeration.

- An escapement constant item (type DDIF$_SFV_ESC_CONSTANT) that specifies the constant measurement to be used as an escapement. This item is encoded as a variable.
- A reset variable item (type DDIF$_SFV_RESET_VARIABLE) that specifies the label of the variable to be reset by the soft value directive. This item is encoded as a string.
- A reset value indicator (type DDIF$_SFV_RESET_VALUE_C) that indicates whether the soft value directive reset value is specified as a variable or constant value. This item is encoded as an expression enumeration.
- A reset value item (type DDIF$_SFV_RESET_VALUE) that specifies the new value of the variable. This item is encoded as a variable.

Table 6–13 lists the items in a soft value directive aggregate and their corresponding encodings.

Table 6–13 Soft Value Directive Aggregate (DDIF$_SFV)

Item Name	Item Encoding
DDIF$_SFV_C	Enumeration
DDIF$_SFV_ESC_RATIO_N	Integer
DDIF$_SFV_ESC_RATIO_D	Integer
DDIF$_SFV_ESC_CONSTANT_C	Measurement enumeration
DDIF$_SFV_ESC_CONSTANT	Variable
DDIF$_SFV_RESET_VARIABLE	String
DDIF$_SFV_RESET_VALUE_C	Expression enumeration
DDIF$_SFV_RESET_VALUE	Variable

6.9 Bézier Curve Content

A cubic Bézier curve is defined by four points. The first set of control points is the first four points in the sequence. Each subsequent set of three points uses the last point of the previous sequence as the first control point in the new sequence.

The Bézier curve content aggregate (type DDIF$_BEZ) contains the following items:

- A flags item (type DDIF$_BEZ_FLAGS) that is used to control the rendition of the curve. This item is encoded as a longword. The flags values are as follows:

DDIF$M_BEZ_DRAW_CURVE	If set, the curve is drawn.
DDIF$M_BEZ_FILL_CURVE	If set, the area within the curve is filled according to the current fill attributes.

DDIF$M_BEZ_CLOSE_CURVE	Determines whether an open or closed curve is drawn. (An open curve whose first and last points are connected by a straight line differs from a closed curve in that a closed curve reuses the first control point as the last control point. A closed cubic curve must consist of at least 6 points.)

The default is DDIF$M_BEZ_DRAW_CURVE.

- A curve path indicator (type DDIF$_BEZ_PATH_C) that specifies whether the layout of the curve is specified as a variable or constant value. This item is encoded as an array of type measurement enumeration.
- A curve path item (type DDIF$_BEZ_PATH) that contains the *x,y* pairs that define the control points of the curve. This item is encoded as an array of type variable.

  The points of the curve are stored in an array in a repeating *x,y*-pair format. For example, if you are storing values in this item, the first value you specify must be the *x* position of the first control point; the second value must be the *y* position of the first control point, and so on. Because these points are stored in an array, you must increment the aggregate index associated with the array each time you read or write a control point. The initial aggregate index value is 0.

  If the layout is frame based, each coordinate is relative to the frame in which it is being rendered. If the layout is path based, each coordinate is relative to the current position on the path.

Table 6–14 lists the items in a Bézier curve aggregate and their corresponding encodings.

Table 6–14 Bézier Curve Aggregate (DDIF$_BEZ)

Item Name	Item Encoding
DDIF$_BEZ_FLAGS	Longword
DDIF$_BEZ_PATH_C	Array of type measurement enumeration
DDIF$_BEZ_PATH	Array of type variable

6.10 Polyline Content

The polyline content aggregate (type DDIF$_LIN) represents polylines, polymarkers, and filled areas. It contains the following items:

- A flags item (type DDIF$_LIN_FLAGS) that is used to control the rendering of the polyline. This item is encoded as a longword. Valid values for this item are as follows:

DDIF$M_LIN_DRAW_POLYLINE	If set, a line is drawn between the specified points; if clear, no line is drawn.

DDIF$M_LIN_FILL_POLYLINE	If set, the area defined by the points is filled; if clear, the area is not filled.
DDIF$M_LIN_DRAW_MARKERS	If set, a marker is placed at each point; if clear, no markers are drawn.
DDIF$M_LIN_REGULAR_POLYGON	If set, the object is a regular polygon.
DDIF$M_LIN_CLOSE_POLYLINE	If set, the last point of the object is connected to the first.
DDIF$M_LIN_ROUNDED_POLYLINE	If set, the line joints of the polyline are rounded.
DDIF$M_LIN_RECTANGULAR_ POLYGON	If set, the polyline represents a rectangle. The polyline must consist of four points. If all four lines must be drawn, the DDIF$M_LIN_CLOSE_ POLYLINE value must also be specified.

- A draw pattern item (type DDIF$_LIN_DRAW_PATTERN) that determines which line segments are drawn. This item is encoded as a bit string.

  Starting from the first bit and the line between the first two points of the object, if the corresponding bit is set, the line is drawn. Otherwise, the line is not drawn, but does limit the fill area.

  The number of bits in the draw pattern does not have to match the number of line segments in the polyline. If the draw pattern contains fewer flags than the object contains line segments, the pattern is repeated. For example, a bit pattern of 1 causes every line to be drawn, and a pattern of 0 suppresses all lines. A pattern of 01 causes every other line to be drawn, beginning with the second. The default is "1"B.

  A draw pattern can be provided even if the DDIF$M_LIN_DRAW_ POLYLINE flag is clear, with the implication that it forms the pattern if the flag is later set.

- A line path indicator (type DDIF$_LIN_PATH_C) that specifies whether the layout of the polyline is specified as a variable or constant value. This item is encoded as an array of type measurement enumeration.

- A line path item (type DDIF$_LIN_PATH) that lists the control points of the polyline. This item is encoded as an array of type variable.

  The points of the polyline are stored in an array in a repeating *x,y*-pair format. For example, if you are storing values in this item, the first value you specify must be the *x* position of the first control point; the second value must be the *y* position of the first control point, and so on. Because these points are stored in an array, you must increment the aggregate index associated with the array each time you read or write a control point. The initial aggregate index value is 0.

  If the layout is frame based, each coordinate is relative to the frame in which it is being rendered. If the layout is path based, each coordinate is relative to the current position on the path.

Table 6–15 lists the items in a polyline aggregate and their corresponding encodings.

Table 6–15 Polyline Aggregate (DDIF$_LIN)

Item Name	Item Encoding
DDIF$_LIN_FLAGS	Longword
DDIF$_LIN_DRAW_PATTERN	Bit string
DDIF$_LIN_PATH_C	Array of type measurement enumeration
DDIF$_LIN_PATH	Array of type variable

6.11 Arc Content

The arc content aggregate (type DDIF$_ARC) contains the following items:

- A flags item (type DDIF$_ARC_FLAGS) that is used to control the rendition of the arc. This item is encoded as a longword. Valid values for this item are as follows:

  DDIF$M_DRAW_ARC — If set, a line is drawn along the arc, rendered as specified by the active line attributes. The line-style pattern should begin at the starting point.

  DDIF$M_FILL_ARC — If set, the arc is filled in the area defined by the arc primitive.

  DDIF$M_PIE_ARC — If set, the boundary for filling/outlining the arc is formed by the arc and the line segments joining the arc endpoints to the center.

  DDIF$M_CLOSE_ARC — If set, and if the DDIF$M_DRAW_ARC flag is set, the outline of the arc is closed. If DDIF$M_PIE_ARC is set, the outline is closed by lines joining the endpoints of the arc with the center. If DDIF$M_PIE_ARC is not set, the outline is closed by a line joining the two arc endpoints.

- An arc center *x* indicator (type DDIF$_ARC_CENTER_X_C) that indicates whether the *x*-coordinate of the center of the arc is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An arc center *x* item (type DDIF$_ARC_CENTER_X) that specifies the *x*-coordinate of the center of the arc. This item is encoded as a variable.
- An arc center *y* indicator (type DDIF$_ARC_CENTER_Y_C) that indicates whether the *y*-coordinate of the center of the arc is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An arc center *y* item (type DDIF$_ARC_CENTER_Y) that specifies the *y* coordinate of the center of the arc. This item is encoded as a variable.

- An arc radius *x* indicator (type DDIF$_ARC_RADIUS_X_C) that indicates whether the *x* radius of the arc is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An arc radius *x* item (type DDIF$_ARC_RADIUS_X) that specifies the distance from the center of the arc to the perimeter of the arc as measured along the *x*-axis. This item is encoded as a variable.
- An arc radius delta *y* indicator (type DDIF$_RADIUS_DELTA_Y_C) that indicates whether the delta *y* radius of the arc is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An arc radius delta *y* item (type DDIF$_ARC_RADIUS_DELTA_Y) that specifies the length difference between the *y* radius and the *x* radius (for example, if the arc is the arc of an ellipse). This item is encoded as a variable. The default value for this item is 0.
- An arc start indicator (type DDIF$_ARC_START_C) that indicates whether the starting angle of the arc is specified as a variable or constant value. This item is encoded as an AngleRef enumeration.
- An arc start item (type DDIF$_ARC_START) that specifies the angle at which the arc is begun. This item is encoded as a variable. The default value for this item is 0.
- An arc extent indicator (type DDIF$_ARC_EXTENT_C) that indicates whether the extent of the arc is specified as a variable or constant value. This item is encoded as an AngleRef enumeration.
- An arc extent item (type DDIF$_ARC_EXTENT) that is added to the arc start angle to determine the end of the arc. This item is encoded as a variable. The default value for this item is 360 degrees.
- An arc rotation indicator (type DDIF$_ARC_ROTATION_C) that indicates whether the angle of rotation of the arc is specified as a variable or constant value. This item is encoded as an AngleRef enumeration.
- An arc rotation item (type DDIF$_ARC_ROTATION) that specifies the angle of rotation of the entire arc relative to the coordinate system. (This item is usually specified for elliptical arcs.) This item is encoded as a variable. The default value for this item is 0 degrees.

Table 6–16 lists the items in an arc content aggregate and their corresponding encodings.

Table 6–16 Arc Content Aggregate (DDIF$_ARC)

Item Name	Item Encoding
DDIF$_ARC_FLAGS	Longword
DDIF$_ARC_CENTER_X_C	Measurement enumeration

(continued on next page)

Table 6–16 (Cont.) Arc Content Aggregate (DDIF$_ARC)

Item Name	Item Encoding
DDIF$_ARC_CENTER_X	Variable
DDIF$_ARC_CENTER_Y_C	Measurement enumeration
DDIF$_ARC_CENTER_Y	Variable
DDIF$_ARC_RADIUS_X_C	Measurement enumeration
DDIF$_ARC_RADIUS_X	Variable
DDIF$_ARC_RADIUS_DELTA_Y_C	Measurement enumeration
DDIF$_ARC_RADIUS_DELTA_Y	Variable
DDIF$_ARC_START_C	AngleRef enumeration
DDIF$_ARC_START	Variable
DDIF$_ARC_EXTENT_C	AngleRef enumeration
DDIF$_ARC_EXTENT	Variable
DDIF$_ARC_ROTATION_C	AngleRef enumeration
DDIF$_ARC_ROTATION	Variable

6.12 Fill Area Set Content

The fill area set content aggregate (type DDIF$_FAS) specifies an arbitrary path that is filled as a unit, or an arbitrary outline. This aggregate contains the following items:

- A fill area set flags item (type DDIF$_FAS_FLAGS) that is used to control the rendition of the fill area. This item is encoded as a longword. Valid values for this item are as follows:

  DDIF$M_FAS_CO_DRAW_BORDER — If set, a line is drawn along the path, using the current line attributes. If the start and end points of the path components are not coincident, a straight line connects the points.

  DDIF$M_FAS_CO_FILL_AREA — If set, the composite area is filled. The fill is performed using the odd winding rule, just as for polylines. (The odd winding rule states that if a ray is drawn from a point to infinity, the origin of the ray is considered inside the area (and hence is filled) if it crosses the area border an odd number of times.) If the start and end points of the path components are not coincident, a straight line connects the points.

  The default value is DDIF$M_FAS_CO_DRAW_BORDER.

- A fill area set path item (type DDIF$_FAS_PATH) that specifies the composite path that constitutes the fill area set. This item is encoded as a sequence of DDIF$_PTH aggregates. For more information on the DDIF$_PTH aggregate, see Section 6.19.

Table 6–17 lists the items in a fill area set content aggregate and their corresponding encodings.

Table 6–17 Fill Area Set Content Aggregate (DDIF$_FAS)

Item Name	Item Encoding
DDIF$_FAS_FLAGS	Longword
DDIF$_FAS_PATH	Sequence of DDIF$_PTH aggregates

6.13 Image Content

Image data is represented as a frame of data within a DDIF document. The origin of a frame is located at the lower left-hand corner of the frame. Any page can contain one or more frames of image data. A frame can contain a single **still image**, or a sequence of **time-varying images** with identical attributes. A frame containing a single image content definition is a still image. A frame containing more than one image content primitive is a time-varying image sequence. Each frame has an attribute that identifies it as either still or time-varying.

Although a DDIF document frame can contain compound data by way of nested frames, a frame of image data is considered atomic. Frames containing any sort of data can be overlaid on a frame of image data to provide some desired effect. For example, you can overlay a frame with text over a frame of image data to create the effect of a border around a picture with text inside. There are no restrictions placed on the inclusion of image frames within other frames of nonimage data.

The image attributes specify the number of pixels in a scan line, and the number of lines, but not the resolution at which the image was scanned. The size of the frame that bounds the image is assumed to represent the original size of the image, and when the image is displayed, it is scaled to fit the bounding box of the frame.

The image content aggregate (type DDIF$_IMG) contains an image content item (type DDIF$_IMG_CONTENT) that specifies the content of the image. This value is encoded as a sequence of DDIF$_IDU aggregates. For more information on the DDIF$_IDU aggregate, see Section 6.18.

It is important to note that the bounding box items of the frame attributes must be respecified in the segment attributes aggregate (type DDIF$_SGA) associated with image content; frame attributes for image content are not inherited from a type definition.

Table 6–18 lists the item in an image content aggregate and its corresponding encoding.

Table 6–18 Image Content Aggregate (DDIF$_IMG)

Item Name	Item Encoding
DDIF$_IMG_CONTENT	Sequence of DDIF$_IDU aggregates

6.14 Content Reference Aggregate

The content reference aggregate (type DDIF$_CRF) enables you to reference a generic content definition. This aggregate contains the following items:

- An optional content reference transformation item (type DDIF$_CRF_TRANSFORM) that specifies a transformation to be applied to all measurements in the referenced content definition. This item is encoded as a sequence of DDIF$_TRN aggregates. (For more information on the DDIF$_TRN aggregate, see Section 6.32.) If a transformation is not supplied, the measurements in the defined content are used unmodified.
- An optional content reference item (type DDIF$_CRF_REFERENCE) that contains the label of the content definition being referenced. This item is encoded as a string.

Table 6–19 lists the items in a content reference aggregate and their corresponding encodings.

Table 6–19 Content Reference Aggregate (DDIF$_CRF)

Item Name	Item Encoding
DDIF$_CRF_TRANSFORM	Sequence of DDIF$_TRN aggregates
DDIF$_CRF_REFERENCE	String

6.15 Restricted Content

In addition to the standard revisable content types, two restricted types are provided: PDL content and private content. Restricted types are limited in terms of interchangeability. In general, PDL content can only be displayed on supporting devices, and is not suitable for revision. Private content is supported only by the creator of the document and perhaps by a limited set of cooperating processors.

6.15.1 External (PDL) Content

The external content aggregate (type DDIF$_EXT) contains the following items:

- An optional direct reference item (type DDIF$_EXT_DIRECT_REFERENCE) that is used to identify the data type (syntax and semantics) of the external element. This item is encoded as an object identifier.

- An optional indirect reference item (type DDIF$_EXT_INDIRECT_REFERENCE). This item is encoded as an integer and is reserved for future standardization.
- An optional data value descriptor (type DDIF$_EXT_DATA_VALUE_DESCRIPTOR) that is a text string describing the external data value to programs and/or users. This item is encoded as a string.
- An encoding indicator (type DDIF$_EXT_ENCODING_C) that indicates the method of encoding of the data value. This item is encoded as an enumeration. Valid values for the encoding indicator are as follows:

DDIF$K_DOCUMENT_ENCODING	Nested document. In this case, the DDIF$_EXT_ENCODING item is encoded as a document root aggregate.
DDIF$K_DDIS_ENCODING	Nested document. In this case, the DDIF$_EXT_ENCODING item uses a DDIS encoding.
DDIF$K_OCTET_ENCODING	Octet-aligned encoding. In this case, the DDIF$_EXT_ENCODING item is encoded as a string.
DDIF$K_ARBITRARY_ENCODING	Arbitrary. In this case, the DDIF$_EXT_ENCODING item is encoded as a bit string.

- An encoding item (type DDIF$_EXT_ENCODING) that specifies the external data value in the specified encoding. This item is encoded as a variable.
- An encoding length item (type DDIF$_EXT_ENCODING_L) that specifies the length (on input) of the encoding. This item is encoded as an integer.

Table 6–20 lists all the items in an external content aggregate and their corresponding encodings.

Table 6–20 External Content Aggregate (DDIF$_EXT)

Item Name	Item Encoding
DDIF$_EXT_DIRECT_REFERENCE	Object identifier
DDIF$_EXT_INDIRECT_REFERENCE	Integer
DDIF$_EXT_DATA_VALUE_DESCRIPTOR	String
DDIF$_EXT_ENCODING_C	Enumeration
DDIF$_EXT_ENCODING	Variable
DDIF$_EXT_ENCODING_L	Integer

6.15.2 Private Content

Private data is defined as compound document semantics that are restricted either to a particular document processing implementation, or to a set of related implementations that support identical private encodings.

There are three places in a compound document where a document processor can escape to private data:

- In the header (for document-wide private indicators)
- In segment attributes (for hierarchical or inheritable data)
- As a content type (for content-like private data or markers)

For example, you can use private data in the following ways:

- As a marker in the document content that indicates the user's last editing position in the document
- As a data element in the header of the document that indicates the menu setups or operation modes that were active at the time the document was written
- To indicate special hyphenation rules that cannot be represented by other means in DDIF
- To specify data that allows the graphics in the document to be edited by a special flowchart editor

The private content aggregate (type DDIF$_PVT) contains the following items:

- A value name item (type DDIF$_PVT_NAME) that uniquely identifies the value. This item is encoded as a string.
- A value data indicator (type DDIF$_PVT_DATA_C) that indicates the type of data that has been named. This item is encoded as an enumeration. Valid values for the data indicator are as follows:

DDIF$K_VALUE_BOOLEAN	Indicates a Boolean value. In this case, the DDIF$_PVT_DATA item is encoded as a type Boolean.
DDIF$K_VALUE_INTEGER	Indicates an integer value. In this case, the DDIF$_PVT_DATA item is encoded as an integer.
DDIF$K_VALUE_TEXT	Indicates a text string value. In this case, the DDIF$_PVT_DATA item is encoded as an array of type character string.
DDIF$K_VALUE_GENERAL	Indicates a stream of bytes in any format. In this case, the DDIF$_PVT_DATA item is encoded as a string.

DDIF$K_VALUE_REFERENCE	Indicates a data value that is a reference to a segment in the document or a segment in another document. In this case, the DDIF$_PVT_DATA item is encoded as a string. For this case, DDIF$_PVT_REFERENCE_ERF_INDEX must also be specified.
DDIF$K_VALUE_LIST	Indicates a list of data values such as the above. In this case, the DDIF$_PVT_DATA item is encoded as a sequence of DDIF$_PVT aggregates. In the nested DDIF$_PVT aggregates, the DDIF$_PVT_NAME item is ignored.
DDIF$K_VALUE_EXTERNAL	Indicates a data value that is represented in a syntax. In this case, the DDIF$_PVT_DATA item is encoded as the handle of an aggregate of type DDIF$_EXT.

- A value data item (type DDIF$_PVT_DATA) that specifies the data value of the specified type. This item is encoded as a variable.
- An external reference index item (type DDIF$_PVT_REFERENCE_ERF_INDEX) that specifies an index into a list of external references. This item is encoded as an integer.

Table 6–21 lists the items in a private content aggregate and their corresponding encodings.

Table 6–21 Private Content Aggregate (DDIF$_PVT)

Item Name	Item Encoding
DDIF$_PVT_NAME	String
DDIF$_PVT_DATA_C	Enumeration
DDIF$_PVT_DATA	Variable
DDIF$_PVT_REFERENCE_ERF_INDEX	Integer

6.16 Layout Galley

The layout galley aggregate (type DDIF$_GLY) lets you describe the shape and attributes of a single galley.

A galley can be used to control the flow of text along a series of parallel paths. These paths are determined by a formatter based on the outline of the galley, the height of the characters on the lines, and other layout parameters such as leading.

Like graphic objects such as lines and curves, galleys are relative to a frame: either the page frame defined by a page layout description, or a floating frame. Also like graphic objects, galleys are imaged in the order in which they are described. Graphic elements can be described and imaged before, after, and between galleys. A galley is not imaged when it is selected for filling with text, but rather in the normal sequence in which objects in the frame are imaged. A page frame and its contents are imaged when the first galley on the page is selected.

The layout galley aggregate contains the following items:

- A galley label item (type DDIF$_GLY_ID) that specifies a label by which the galley can be referenced. This item is encoded as a string.
- A lower left corner *x* position indicator (type DDIF$_GLY_BOUNDING_BOX_LL_X_C) that indicates whether the lower left corner *x*-coordinate is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A lower left corner *x* position item (type DDIF$_GLY_BOUNDING_BOX_LL_X) that specifies the *x*-coordinate of the lower left corner of the galley. This item is encoded as a variable.
- A lower left corner *y* position indicator (type DDIF$_GLY_BOUNDING_BOX_LL_Y_C) that indicates whether the lower left corner *y*-coordinate is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A lower left corner *y* position item (type DDIF$_GLY_BOUNDING_BOX_LL_Y) that specifies the *y*-coordinate of the lower left corner of the galley. This item is encoded as a variable.
- An upper right corner *x* position indicator (type DDIF$_GLY_BOUNDING_BOX_UR_X_C) that indicates whether the upper right corner *x*-coordinate is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An upper right corner *x* position item (type DDIF$_GLY_BOUNDING_BOX_UR_X) that specifies the *x*-coordinate of the upper right corner of the galley. This item is encoded as a variable.
- An upper right corner *y* position indicator (type DDIF$_GLY_BOUNDING_BOX_UR_Y_C) that indicates whether the upper right corner *y*-coordinate is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An upper right corner *y* position item (type DDIF$_GLY_BOUNDING_BOX_UR_Y) that specifies the *y*-coordinate of the upper right corner of the galley. This item is encoded as a variable.
- An optional galley outline item (type DDIF$_GLY_OUTLINE) that specifies the path to which content within the galley is formatted. This item is encoded as a sequence of DDIF$_PTH aggregates. (For more information on the DDIF$_PTH aggregate, see Section 6.19.) The outline is constrained to fit within the bounding box, and defaults to the rectangle defined as the bounding box. Content is formatted inside the path, where the inside is determined by the odd winding rule. (The odd winding rule states that, if a ray is drawn from a point to infinity, the origin of the ray is considered inside the area (and hence will be filled) if it crosses the area border an odd number of times.)
- An optional layout galley flags item (type DDIF$_GLY_FLAGS) that controls the display of the galley or its content. This item is encoded as a longword. Valid values are as follows:

DDIF$M_GLY_VERTICAL_ALIGN	The elements in the galley are adjusted so that the vertical space in the galley is completely used.
DDIF$M_GLY_BORDER	A border is drawn around the outline of the galley.
DDIF$M_GLY_AUTOCONNECT	If text overflows the galley during layout, it automatically flows into the successor galley. If the successor is a generic galley (is on a generic page) then an instance of that page will be created.
DDIF$M_GLY_BACKGROUND_FILL	The current fill pattern or color is used to fill the galley before the text that flows into the galley is imaged.

- An optional galley streams item (type DDIF$_GLY_STREAMS) that specifies the content streams that can appear in the galley. This item is encoded as an array of type string.
- A galley successor indicator (type DDIF$_GLY_SUCCESSOR_C) that indicates the type of galley to be used when text overflows. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_GENERIC_GALLEY	Indicates a galley on a page in the generic layout. In this case, the DDIF$_GLY_SUCCESSOR item is encoded as a string.
DDIF$K_SPECIFIC_GALLEY	Indicates a galley on a page in specific layout. In this case, the DDIF$_GLY_SUCCESSOR item is encoded as a string.
DDIF$K_NO_SUCCESSOR_GALLEY	Indicates that there is no successor galley and overflow text is not displayed. In this case, you should not specify the DDIF$_GLY_SUCCESSOR item.

- A galley successor item (type DDIF$_GLY_SUCCESSOR) that specifies the galley used when text overflows. This item is encoded as a variable.

Table 6–22 lists all the items in the layout galley aggregate and their corresponding encodings.

Table 6–22 Layout Galley Aggregate (DDIF$_GLY)

Item Name	Item Encoding
DDIF$_GLY_ID	String
DDIF$_GLY_BOUNDING_BOX_LL_X_C	Measurement enumeration
DDIF$_GLY_BOUNDING_BOX_LL_X	Variable
DDIF$_GLY_BOUNDING_BOX_LL_Y_C	Measurement enumeration

(continued on next page)

Table 6–22 (Cont.) Layout Galley Aggregate (DDIF$_GLY)

Item Name	Item Encoding
DDIF$_GLY_BOUNDING_BOX_LL_Y	Variable
DDIF$_GLY_BOUNDING_BOX_UR_X_C	Measurement enumeration
DDIF$_GLY_BOUNDING_BOX_UR_X	Variable
DDIF$_GLY_BOUNDING_BOX_UR_Y_C	Measurement enumeration
DDIF$_GLY_BOUNDING_BOX_UR_Y	Variable
DDIF$_GLY_OUTLINE	Sequence of DDIF$_PTH aggregates
DDIF$_GLY_FLAGS	Longword
DDIF$_GLY_STREAMS	Array of type string
DDIF$_GLY_SUCCESSOR_C	Enumeration
DDIF$_GLY_SUCCESSOR	Variable

6.17 External Reference

An external reference element describes a source of data that is outside the document. It does so by specifying the data syntax and location of the external reference element. An external reference aggregate (type DDIF$_ERF) contains the following items:

- A reference data type item (type DDIF$_ERF_DATA_TYPE) that identifies the data type of the external data object. This item is encoded as an object identifier. An object identifier is specified as an array of seven longwords. Table 6–23 lists each supported object identifier, as well as the seven longword data values used to specify that object identifier type. Note that the values must be written to the array in the order in which they appear in Table 6–23 (from left to right).

Table 6–23 Object Identifier Table

Array Values							Object Identifier Type
1	3	12	1011	1	3	1	DIGITAL Document Interchange Format (DDIF)
1	3	12	1011	1	3	2	Data Object Transport Syntax (DOTS)
1	3	12	1011	1	3	4	ASCII-text data stream
1	3	12	1011	1	3	5	Application-dependent data
1	3	12	1011	1	3	6	PostScript

Object identifiers are used by the CDA Toolkit to denote the assigned semantics of stored DDIF files. These data formats are uniquely identifiable because an object identifier consists of a hierarchy of subidentifiers that designate those groups that have registered the subregistry or the data type.

- A reference descriptor item (type DDIF$_ERF_DESCRIPTOR) that provides a human-readable description of the data type. This item is encoded as an array of type character string.
- A reference label item (type DDIF$_ERF_LABEL) that provides the label by which the user or the system identifies the data object (that is, the file specification of the external file). This item is encoded as a character string.
- A storage item (type DDIF$_ERF_LABEL_TYPE) that contains a tag that identifies the type of storage system in which the external reference is located. This item is encoded as a string with *add-info*. The following table lists the values for *add-info* and the corresponding string values.

DDIF$K_PRIVATE_LABEL_TYPE	The label is a private label. In this case, the string can be any user-specified string.
DDIF$K_RMS_LABEL_TYPE	The label is an RMS file specification. In this case, the string must be "$RMS".
DDIF$K_UTX_LABEL_TYPE	The label is an ULTRIX file specification. In this case, the string must be "$UTX".
DDIF$K_MDS_LABEL_TYPE	The label is an MS-DOS or OS/2 file specification. In this case, the string must be "$MDS".
DDIF$K_STYLE_LABEL_TYPE	The label type is a style-guide file specification. In this case, the string must be "$STYLE".

- A control item (type DDIF$_ERF_CONTROL) that specifies how the referenced data object is treated when the document is transferred from one system to another. This item is encoded as an enumeration. Valid values for this item are as follows:

DDIF$K_COPY_REFERENCE	The referenced data object is transmitted along with the document, and is stored on the receiving system.
DDIF$K_NO_COPY_REFERENCE	The referenced data is not transmitted with the document.

Table 6–24 lists the items in an external reference aggregate and their corresponding encodings.

Table 6–24 External Reference Aggregate (DDIF$_ERF)

Item Name	Item Encoding
DDIF$_ERF_DATA_TYPE	Object identifier
DDIF$_ERF_DESCRIPTOR	Array of type character string
DDIF$_ERF_LABEL	Character string
DDIF$_ERF_LABEL_TYPE	String with *add-info*
DDIF$_ERF_CONTROL	Enumeration

6.18 Image Data Unit

The image data unit aggregate (type DDIF$_IDU) describes image data in terms of its image coding attributes and the actual image data. This aggregate contains the following items:

- An optional private coding attributes item (type DDIF$_IDU_PRIVATE_CODING_ATTR) that provides for the addition of application-private image coding attributes. This item is encoded as a sequence of DDIF$_PVT aggregates. Data placed here can be of any type and any structure. For more information on the DDIF$_PVT aggregate, see Section 6.15.2.
- A pixels-per-line item (type DDIF$_IDU_PIXELS_PER_LINE) that specifies the total number of pixels per scanline. This item is encoded as an integer. Note that the pixels-per-line item does not necessarily represent the total number of bits per scanline.
- A number-of-lines item (type DDIF$_IDU_NUMBER_OF_LINES) that specifies the total number of scanlines in an image. This item is encoded as an integer.
- A compression type item (type DDIF$_IDU_COMPRESSION_TYPE) that indicates the compression scheme used to encode a particular plane of image data. This item is encoded as an enumeration. Valid values for this item are as follows:

DDIF$K_PRIVATE_COMPRESSION	Private compression scheme
DDIF$K_PCM_COMPRESSION	Raw bitmap
DDIF$K_G31D_COMPRESSION	Consultative Committee on International Telephony and Telegraphy (CCITT) Group 3 1-dimensional
DDIF$K_G32D_COMPRESSION	CCITT Group 3 2-dimensional
DDIF$K_G42D_COMPRESSION	CCITT Group 4 2-dimensional

  DDIF$K_PCM_COMPRESSION is the default.
- An optional compression parameters item (type DDIF$_IDU_COMPRESSION_PARAMS) that contains the parameters required for the specified compression. This item is encoded as a sequence of DDIF$_PVT aggregates. For more information on the DDIF$_PVT aggregate, see Section 6.15.2.
- A data offset item (type DDIF$_IDU_DATA_OFFSET) that specifies the offset (in bits) from the start of the octet string to the first bit of image data. This item is encoded as an integer. The default for the data offset item is 0.
- An optional pixel stride item (type DDIF$_IDU_PIXEL_STRIDE) that specifies the difference in bit addresses between successive pixels. This item is encoded as an integer.

  Pixel stride is typically equal to the number of bits per pixel stored in a particular data plane. If pixel alignment requires fill bits between pixels, the difference between this value and the number of bits per pixel per component equals the fill value.

- An optional scanline stride item (type DDIF$_IDU_SCANLINE_STRIDE) that specifies the difference in bit addresses between the starting bits of successive scanlines. This item is encoded as an integer.

  Scanline stride is typically equal to the number of bits (not pixels) per scanline. If scanline alignment requires fill bits between scanlines, the difference between scanline stride and the number of bits per scanline equals the fill value. In most cases, when image data is compressed, scanline stride has little meaning and is not present.

- A pixel order item (type DDIF$_IDU_PIXEL_ORDER) that specifies the order in which pixel data is stored within each byte. This item is encoded as an enumeration. Valid values for this item are as follows:

DDIF$K_STANDARD_PIXEL_ORDER	Indicates standard pixel order
DDIF$K_REVERSE_PIXEL_ORDER	Indicates reverse pixel order

  The default value is DDIF$K_STANDARD_PIXEL_ORDER.

- An optional plane-bits-per-pixel item (type DDIF$_IDU_BITS_PER_PIXEL) that indicates the total number of bits per pixel. This item is encoded as an integer. This value also represents the sum of the number of bits per component for all components. For bitonal images, the plane-bits-per-pixel item always has a value of 1, and is therefore omitted.

- A plane data item (type DDIF$_IDU_PLANE_DATA) that specifies the actual data. This item is encoded as a string.

Table 6–25 lists all the items in an image data unit aggregate and their corresponding encodings.

Table 6–25 Image Data Unit Aggregate (DDIF$_IDU)

Item Name	Item Encoding
DDIF$_IDU_PRIVATE_CODING_ATTR	Sequence of DDIF$_PVT aggregates
DDIF$_IDU_PIXELS_PER_LINE	Integer
DDIF$_IDU_NUMBER_OF_LINES	Integer
DDIF$_IDU_COMPRESSION_TYPE	Enumeration
DDIF$_IDU_COMPRESSION_PARAMS	Sequence of DDIF$_PVT aggregates
DDIF$_IDU_DATA_OFFSET	Integer
DDIF$_IDU_PIXEL_STRIDE	Integer
DDIF$_IDU_SCANLINE_STRIDE	Integer
DDIF$_IDU_PIXEL_ORDER	Enumeration
DDIF$_IDU_BITS_PER_PIXEL	Integer
DDIF$_IDU_PLANE_DATA	String

6.19 Composite Path

A composite path type defines an arbitrary path as a sequence of other path types (polylines, arcs, cubic Béziers, and other composite paths). The composite path aggregate (type DDIF$_PTH) contains the following items:

- A path indicator (type DDIF$_PTH_C) that indicates the type of path component being defined. This item is encoded as an enumeration. Valid values for this item are as follows:

DDIF$K_PATH_LINE	Indicates a polyline component of the path. If you specify this value, you must supply values for the items DDIF$_PTH_LIN_PATH_C through DDIF$_PTH_LIN_PATH.
DDIF$K_PATH_BEZIER	Indicates a cubic Bézier component of the path. If you specify this value, you must supply values for the items DDIF$_PTH_BEZ_PATH_C through DDIF$_PTH_BEZ_PATH.
DDIF$K_PATH_ARC	Indicates an arc component of the path. If you specify this value, you must supply values for the items DDIF$_PTH_ARC_CENTER_X_C through DDIF$_PTH_ARC_ROTATION.
DDIF$K_PATH_REFERENCE	Indicates a reference to a defined component of the path. If you specify this value, you must supply a value for the item DDIF$_PTH_REFERENCE.

- A line path indicator (type DDIF$_PTH_LIN_PATH_C) that specifies whether the layout of the polyline is specified as a variable or constant value. This item is encoded as an array of type measurement enumeration.
- A line path item (type DDIF$_PTH_LIN_PATH) that lists the control points of the polyline. This item is encoded as an array of type variable.

  The points of the polyline are stored in an array in a repeating *x,y*-pair format. For example, if you are storing values in this item, the first value you specify must be the *x* position of the first control point; the second value must be the *y* position of the first control point, and so on. Because these points are stored in an array, you must increment the aggregate index associated with the array each time you read or write a control point. The initial aggregate index value is 0.

  Note that each coordinate is relative to the frame in which it is being rendered.
- A curve path indicator (type DDIF$_PTH_BEZ_PATH_C) that specifies whether the layout of the curve is specified as a variable or constant value. This item is encoded as an array of type measurement enumeration.
- A curve path item (type DDIF$_PTH_BEZ_PATH) that contains the *x,y*-pairs that define the control points of the curve. This item is encoded as an array of type variable.

The points of the curve are stored in an array in a repeating *x,y*-pair format. For example, if you are storing values in this item, the first value you specify must be the *x* position of the first control point; the second value must be the *y* position of the first control point, and so on. Because these points are stored in an array, you must increment the aggregate index associated with the array each time you read or write a control point. The initial aggregate index value is 0.

- An arc center *x* indicator (type DDIF$_PTH_ARC_CENTER_X_C) that indicates whether the *x*-coordinate of the center of the circle of which this arc is a part is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An arc center *x* item (type DDIF$_PTH_ARC_CENTER_X) that specifies the *x*-coordinate of the center of the circle of which this arc is a part. This item is encoded as a variable.
- An arc center *y* indicator (type DDIF$_PTH_ARC_CENTER_Y_C) that indicates whether the *y*-coordinate of the center of the circle of which this arc is a part is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An arc center *y* item (type DDIF$_PTH_ARC_CENTER_Y) that specifies the *y*-coordinate of the center of the circle of which this arc is a part. This item is encoded as a variable.
- An arc radius *x* indicator (type DDIF$_PTH_ARC_RADIUS_X_C) that indicates whether the *x*-radius of the arc is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An arc radius *x* item (type DDIF$_PTH_ARC_RADIUS_X) that specifies the distance from the center of the arc to the perimeter of the arc as measured along the *x*-axis. This item is encoded as a variable.
- An arc radius delta *y* indicator (type DDIF$_PTH_ARC_RADIUS_DELTA_Y_C) that indicates whether the delta *y*-radius of the arc is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An arc radius delta *y* item (type DDIF$_PTH_ARC_RADIUS_DELTA_Y) that specifies the length difference between the *y*-radius and the *x*-radius (for example, if the arc is the arc of an ellipse). This item is encoded as a variable. The default value for this item is 0.
- An arc start indicator (type DDIF$_PTH_ARC_START_C) that indicates whether the starting angle of the arc is specified as a variable or constant value. This item is encoded as an AngleRef enumeration.
- An arc start item (type DDIF$_PTH_ARC_START) that specifies the angle at which the arc is begun. This item is encoded as a variable. The default value for this item is 0.
- An arc extent indicator (type DDIF$_PTH_ARC_EXTENT_C) that indicates whether the extent of the arc is specified as a variable or constant value. This item is encoded as an AngleRef enumeration.

- An arc extent item (type DDIF$_PTH_ARC_EXTENT) that is added to the arc start angle to determine the end of the arc. This item is encoded as a variable. The default value for this item is 360 degrees.
- An arc rotation indicator (type DDIF$_PTH_ARC_ROTATION_C) that indicates whether the angle of rotation of the arc is specified as a variable or as a constant value. This item is encoded as an AngleRef enumeration.
- An arc rotation item (type DDIF$_PTH_ARC_ROTATION) that specifies the angle of rotation of the entire arc relative to the coordinate system. (This item is usually specified for elliptical arcs.) This item is encoded as a variable. The default value for this item is 0 degrees.
- A path reference item (type DDIF$_PTH_REFERENCE) that provides a reference to a defined component of the path, which is itself a composite path. This item is encoded as an integer.

Table 6–26 lists all of the items in a composite path aggregate and their corresponding encodings.

Table 6–26 Composite Path Aggregate (DDIF$_PTH)

Item Name	Item Encoding
DDIF$_PTH_C	Enumeration
DDIF$_PTH_LIN_PATH_C	Array of type measurement enumeration
DDIF$_PTH_LIN_PATH	Array of type variable
DDIF$_PTH_BEZ_PATH_C	Array of type measurement enumeration
DDIF$_PTH_BEZ_PATH	Array of type variable
DDIF$_PTH_ARC_CENTER_X_C	Measurement enumeration
DDIF$_PTH_ARC_CENTER_X	Variable
DDIF$_PTH_ARC_CENTER_Y_C	Measurement enumeration
DDIF$_PTH_ARC_CENTER_Y	Variable
DDIF$_PTH_ARC_RADIUS_X_C	Measurement enumeration
DDIF$_PTH_ARC_RADIUS_X	Variable
DDIF$_PTH_ARC_RADIUS_DELTA_Y_C	Measurement enumeration
DDIF$_PTH_ARC_RADIUS_DELTA_Y	Variable
DDIF$_PTH_ARC_START_C	AngleRef enumeration
DDIF$_PTH_ARC_START	Variable
DDIF$_PTH_ARC_EXTENT_C	AngleRef enumeration
DDIF$_PTH_ARC_EXTENT	Variable
DDIF$_PTH_ARC_ROTATION_C	AngleRef enumeration
DDIF$_PTH_ARC_ROTATION	Variable
DDIF$_PTH_REFERENCE	Integer

6.20 Segment Attributes

The segment attributes aggregate (type DDIF$_SGA) defines the presentation and processing characteristics of a segment of document content. The items in this aggregate can be broken down into the following logical groups:

- General segment attribute items
- Computed content attribute items
- Structure items
- A language attribute item
- Legend items
- Measurement items
- An alternate presentation item
- Layout items
- A font definition item
- A pattern definition item
- A path definition item
- A line-style definition item
- A content definition item
- A type definition item
- Text attribute items
- Line attribute items
- Marker attribute items
- A galley attribute item
- Image attribute items
- Image space items
- Frame items
- An item-change-list item

Each of these items, or groups of items, is discussed in the following sections. Where appropriate, default or initial values are specified. Section 6.20.23 lists all of the items in the DDIF$_SGA aggregate and their corresponding encodings.

6.20.1 General Segment Attributes

The segment attributes aggregate contains the following items that specify general segment attributes:

- An optional private attributes item (type DDIF$_SGA_PRIVATE_DATA) that specifies any product-specific attributes for the segment. This item is encoded as a sequence of DDIF$_PVT aggregates. (For

more information on the DDIF$_PVT aggregate, see Section 6.15.2.) No initial private attributes are defined.

- An optional content streams item (type DDIF$_SGA_CONTENT_STREAMS) that specifies the content streams to which the segment content belongs. This item is encoded as an array of type string. The initial content stream is "$DB", which denotes the document body.
- An optional content category item (type DDIF$_SGA_CONTENT_CATEGORY) that indicates the category of content, such as text ($T), graphics ($2D), or image ($I), to which the content of the segment belongs. This item is encoded as a string with *add-info*, where *add-info* can take the following values:

DDIF$K_PRIVATE_CATEGORY	The content is nonstandard or was standardized after the release of the Toolkit.
DDIF$K_I_CATEGORY	The content is of the image ($I) category.
DDIF$K_2D_CATEGORY	The content is of the graphics ($2D) category.
DDIF$K_T_CATEGORY	The content is of the text ($T) category.
DDIF$K_TBL_CATEGORY	The content is of the table ($TBL) category.
DDIF$K_PDL_CATEGORY	The content is of the page description language ($PDL) category.

The initial value is DDIF$K_T_CATEGORY, meaning that the content category is text ($T).

- An optional segment tags item (type DDIF$_SGA_SEGMENT_TAGS) that specifies tags that denote the processing characteristics of the content. This item is encoded as an array of type string with *add-info*, where *add-info* can take the following values:

DDIF$K_PRIVATE_TAG	The segment tag is a nonstandard tag.
DDIF$K_CRF_TAG	The segment tag is a cross-reference ($CRF) tag.
DDIF$K_F_TAG	The segment tag is a figure ($F) tag.
DDIF$K_P_TAG	The segment tag is a paragraph ($P) tag.
DDIF$K_S_TAG	The segment tag is a section ($S) tag.
DDIF$K_I_TAG	The segment tag is an index ($I) tag.
DDIF$K_E_TAG	The segment tag is an emphasis ($E) tag.
DDIF$K_L_TAG	The segment tag is a list ($L) tag.
DDIF$K_LE_TAG	The segment tag is a list element ($LE) tag.
DDIF$K_LIT_TAG	The segment tag is a literal ($LIT) tag.
DDIF$K_FN_TAG	The segment tag is a footnote ($FN) tag.
DDIF$K_AN_TAG	The segment tag is an annotation ($AN) tag.
DDIF$K_LBL_TAG	The segment tag is a label ($LBL) tag.
DDIF$K_TTL_TAG	The segment tag is a title ($TTL) tag.
DDIF$K_GRP_TAG	The segment tag is a group member ($GRP) tag.
DDIF$K_GO_TAG	The segment tag is a graphic object ($GO) tag.

Initially, there are no segment tags specified.

- An optional segment binding item (type DDIF$_SGA_BINDING_DEFNS) that lists the variables bound to the segment. This item is encoded as a sequence of DDIF$_SGB aggregates. (For more information on the DDIF$_SGB aggregate, see Section 6.26.) Initially there are no segment bindings.

6.20.2 Computed Content Attributes

The segment attributes aggregate contains items used to control computed content attributes. The computed content attributes are specified by first selecting the type of computed content, and then specifying the appropriate information for that type.

To select the computed content type, a computed content indicator (type DDIF$_SGA_COMPUTE_C) is used. This value is encoded as an enumeration. Valid values are as follows:

DDIF$K_COPY_COMPUTE	Indicates that the content originates from another segment in this document, or an external document, and that the content is updated only at the user's request. If you specify this computed content type, you must supply values for the items DDIF$_SGA_CPTCPY_TARGET and DDIF$_SGA_CPTCPY_ERF_INDEX.
DDIF$K_REMOTE_COMPUTE	Indicates that the content originates from another segment in this document, or an external document, and that the content is updated every time it is displayed. If you specify this computed content type, you must supply values for the items DDIF$_SGA_CPTCPY_TARGET and DDIF$_SGA_CPTCPY_ERF_INDEX.
DDIF$K_VARIABLE_COMPUTE	Indicates the content source as the current value that is bound to a variable by this segment or in some parent segment. If you specify this computed content type, you must supply a value for the item DDIF$_SGA_CPTVAR_VARIABLE.
DDIF$K_XREF_COMPUTE	Indicates the content source as the current value that is bound to a variable at the indicated target segment. If you specify this computed content type, you must supply values for the items DDIF$_SGA_CPTXRF_TARGET through DDIF$_SGA_CPTXRF_VARIABLE.
DDIF$K_FUNCTION_COMPUTE	Indicates the content source as the result of some external process applied to parameters. If you specify this computed content type, you must supply values for the items DDIF$_SGA_CPTFNC_NAME and DDIF$_SGA_CPTFNC_PARAMETERS.

Each of these computed content types is discussed in the following sections, along with its corresponding items.

6.20.2.1 Copied and Remote Computed Content

The copied computed content (selected by specifying DDIF$_SGA_COMPUTE_C as DDIF$K_COPY_COMPUTE or DDIF$K_REMOTE_COMPUTE) is specified using the following items:

- A reference target item (type DDIF$_SGA_CPTCPY_TARGET) that indicates the label of the segment being referenced. This item is encoded as a string. If this item is not specified, the entire document is being referenced.
- A reference index item (type DDIF$_SGA_CPTCPY_ERF_INDEX) that specifies an index into a list of external references stored in the document header. This item is encoded as an integer. If this item is not specified, the reference is to the current document.

In the case of remote computed content, the same aggregate items apply. The difference is that, for copied computed content, the content of the segment is updated only at the user's request. In the case of remote content, the content of the segment is updated when the document is received.

6.20.2.2 Variable Computed Content

The variable computed content (selected by specifying DDIF$_SGA_COMPUTE_C as DDIF$K_VARIABLE_COMPUTE) contains a variable item (type DDIF$_CPTVAR_VARIABLE) that specifies the name of the variable. This item is encoded as a string.

6.20.2.3 Cross-Reference Computed Content

The cross-reference computed content (selected by specifying DDIF$_SGA_COMPUTE_C as DDIF$K_XREF_COMPUTE) contains the following items:

- A cross-reference target segment label (type DDIF$_SGA_CPTXRF_TARGET) that specifies the label by which the target segment is referenced. This item is encoded as a string. If you do not specify a target segment label, the document root segment is referenced.
- A cross-reference index item (type DDIF$_SGA_CPTXRF_ERF_INDEX) that specifies an index into a list of external references stored in the document header. This item is encoded as an integer. If you do not specify a value for this item, the current document is referenced.
- A cross-reference variable label (type DDIF$_SGA_CPTXRF_VARIABLE) that specifies the name of the variable containing the value being referenced. This item is encoded as a string.

6.20.2.4 Function Computed Content

The function computed content (selected by specifying DDIF$_SGA_COMPUTE_C as DDIF$K_FUNCTION_COMPUTE) contains the following items:

- A function name item (type DDIF$_SGA_CPTFNC_NAME) that specifies the name of the function, which is used in conjunction with user-preference information to uniquely identify a program that is to

be invoked with the indicated parameters. This item is encoded as a string.

- A function parameters item (type DDIF$_SGA_CPTFNC_PARAMETERS) that indicates the sequence of parameters required by the function. This item is encoded as a sequence of DDIF$_PVT aggregates. For more information on the DDIF$_PVT aggregate, see Section 6.15.2.

6.20.3 Structure Attributes

The structure attributes specify the legal logical structure of references to segment type definitions within the segment. They describe a set of constraints placed on the ordering, the grouping, and the number of segments with type references. The structure description is initially absent — all combinations of reference are valid.

The structure attributes are specified using the following items:

- A structure description indicator (type DDIF$_SGA_STRUCTURE_DESC_C) that specifies the type of legal logical structure. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_SEQUENCE_STRUCTURE	Indicates a sequence of element occurrences that are constrained to occur in the order specified. In this case, the DDIF$_SGA_STRUCTURE_DESC item is encoded as a sequence of DDIF$_OCC aggregates.
DDIF$K_SET_STRUCTURE	Indicates a set of element occurrences that are *not* constrained with respect to order. In this case, the DDIF$_SGA_STRUCTURE_DESC item is encoded as a sequence of DDIF$_OCC aggregates.
DDIF$K_CHOICE_STRUCTURE	Indicates a group of element occurrences from which only one can be selected. In this case, the DDIF$_SGA_STRUCTURE_DESC item is encoded as a sequence of DDIF$_OCC aggregates.

- A structure description item (type DDIF$_SGA_STRUCTURE_DESC) that specifies the structure itself. This item is encoded as a sequence of DDIF$_OCC aggregates, regardless of which structure is selected using the DDIF$_SGA_STRUCTURE_DESC_C item. (For more information on the DDIF$_OCC aggregate, see Section 6.29.)

6.20.4 Language

The segment attributes aggregate contains an optional language item (type DDIF$_SGA_LANGUAGE) that defines the natural or synthetic (programming) language of text in the segment. This item is encoded as an integer.

The language does not imply text direction or formatting conventions, as these are expressed by presentation and layout attributes. Instead, the language is used to select language tools such as spelling checkers. An initial language is not specified.

6.20.5 Legend

The legend attributes describe the world coordinate system for the content of a segment. Legend units do not affect the rendition of document content. Instead, they indicate the scale of an illustration. There are three legend attribute items:

- A legend unit numerator item (type DDIF$_SGA_LEGEND_UNIT_N) that specifies the magnitude of the ratio of the user coordinate system to the document coordinate system. This item is encoded as an integer. The default value of the ratio is 1:1.
- A legend unit denominator item (type DDIF$_SGA_LEGEND_UNIT_D) that specifies the units of precision used in the ratio. This item is encoded as an integer. The default value for the units of precision is 100.
- A legend unit name item (type DDIF$_SGA_LEGEND_UNIT_NAME) that specifies the name of the user coordinate system. This item is encoded as an array of type character string and has an initial value of inches.

6.20.6 Measurement

The optional measurement attributes describe the coordinate system used within the segment. Measurement units always specify the number of units per inch, regardless of the nesting of segments with measurement unit declarations. The measurement attribute items specify the precision of measurements, rather than the scale of measurements. Note that measurement units specified in specific attributes are in effect for the measurements specified in subsequent attributes.

There are two measurement attribute items:

- A units per measurement item (type DDIF$_SGA_UNITS_PER_MEASURE) that specifies the number of units per inch. This item is encoded as an integer and has an initial value of 1200.
- A unit name item (type DDIF$_SGA_UNIT_NAME) that specifies the name of the measurement system. This item is encoded as an array of type character string and has an initial value of "BMU". The BMU is a Basic Measurement Unit that is a standard unit of measure used in DDIF and equal to 1/1200th of an inch.

6.20.7 Alternate Presentation

The optional alternate presentation item (type DDIF$_SGA_ALT_PRESENTATION) contains a string that can be presented to the user when the content of the segment cannot be displayed. This item is encoded as an array of type character string.

This is an optional string for use with the application's error message under that particular condition. This string is initially absent.

6.20.8 Layout

Layout defines how a text processor images characters along paths. DDIF defines four mechanisms for describing the layout path of text:

1 **Galley-based layout** describes the flow of text among galleys (columns and pages). The parameters used to describe galley-based layout include layout blocks, margins, page sizes, external hyphenation libraries, widow and orphan penalties, and user-specified layout directives such as *new-page*.

In galley-based layout, the location of each successive path is determined algorithmically, but the algorithm may require several passes in order to optimize white space or arrange an illustration close to its referencing text.

Layout of text content in the Text ($T) content category is always galley based. Positional graphics text is usually path based.

2 **Path-based layout** describes the flow of text along a path. This path can be a straight line, a series of line segments, or a curve. Along the path, characters have an orientation with respect to the path itself or with respect to the frame in which they are imaged. For example, characters can be tangent to the path, or upright with respect to the frame. Path-based layout is restricted to the Graphics ($2D) content category.

While segments that specify layout paths are not normally nested within other segments that specify a layout path, such a situation has a defined behavior: text within a segment is placed on the current path. At the end of a nested segment, the previous path is restored.

3 **Frame-based layout** describes the position of each unit of text. Note that the origin of the frame is located at the lower left-hand corner of the frame. Frame-based layout requires that the text unit be located in a subframe, within which one of the above layout methods is used.

Frame-based layout is also the normal layout for graphics objects.

4 **Positional layout** describes the position of the text relative to the current baseline.

The segment attributes aggregate contains items that enable you to specify the layout of content. The layout of the content is described by first selecting the type of layout and then specifying the appropriate information for that type. To select the layout type, an optional layout

indicator (type DDIF$_SGA_LAYOUT_C) is used. This value is encoded as an enumeration. Valid values are as follows:

DDIF$K_GALLEY_LAYOUT	Indicates text laid out in galleys. If you specify this layout type, you must supply values for the items DDIF$_SGA_LAYGLY_WRAP and DDIF$_SGA_LAYGLY_LAYOUT.
DDIF$K_PATH_LAYOUT	Defines a path along which all strings in the segment are imaged. If you specify this layout type, you must supply values for the items DDIF$_SGA_LAYPTH_PATH through DDIF$_SGA_LAYPTH_V_ALIGN.
DDIF$K_RELATIVE_LAYOUT	Indicates that the text is positioned relative to the frame defined by the current segment attributes or those of a parent segment. If you specify this layout type, you must supply values for the items DDIF$_SGA_LAYREL_H_RATIO_N through DDIF$_SGA_LAYREL_V_CONSTANT.
DDIF$K_POSITION_LAYOUT	Specifies the position of the segment relative to the current baseline. If you specify this layout type, you must supply a value for the item DDIF$_SGA_LAYPOS_TEXT_POSITION.

6.20.8.1 Galley-Based Layout

The galley-based layout (selected by specifying DDIF$_SGA_LAYOUT_C as DDIF$K_GALLEY_LAYOUT) is specified using the following items:

- An optional wrap attributes item (type DDIF$_SGA_LAYGLY_WRAP) that indicates the wrap attributes of the galley layout. This item is encoded as the handle of a DDIF$_LW1 aggregate. For more information on the DDIF$_LW1 aggregate, see Section 6.35.
- An optional galley layout item (type DDIF$_SGA_LAYGLY_LAYOUT) that specifies the general layout attributes. This item is encoded as the handle of a DDIF$_LL1 aggregate. For more information on the DDIF$_LL1 aggregate, see Section 6.36.

6.20.8.2 Path-Based Layout

The path-based layout (selected by specifying DDIF$_SGA_LAYOUT_C as DDIF$K_PATH_LAYOUT) is specified using the following items:

- A layout path item (type DDIF$_SGA_LAYPTH_PATH) that identifies the path along which strings are imaged. This item is encoded as a sequence of DDIF$_PTH aggregates. For more information on the DDIF$_PTH aggregate, see Section 6.19.
- A layout format item (type DDIF$_SGA_LAYPTH_FORMAT) that specifies the format of text strings along the string path. The start and end points of the path define the end points for justification. This item is encoded as an enumeration and can accept any of the following values:

DDIF$K_FMT_FLUSH_PATH_BEGIN	The first character is imaged at the start of the text path, and successive characters are imaged at successive positions determined by the escapement of the characters imaged. If the string layout path is shorter than the text string in this case, the path is extended tangent to the slope at the end of the path from the end of the path to the frame clipping outline.
DDIF$K_FMT_CENTER_OF_PATH	The length of text strings, as given by the sum of the character escapements, is subtracted from the length of the path; the remaining space is evenly distributed between the first character and the start of the path, and the last character and the end of the path. If the string layout path is shorter than the text string in this case, the text is forced onto the path by reducing the escapement of the characters in the string.
DDIF$K_FMT_FLUSH_PATH_END	The text string is imaged such that the right alignment point of the last character is aligned with the end of the text string when normal escapement is applied. If the string layout path is shorter than the text string in this case, the path is extended tangent to the beginning of the path, from the beginning of the path to the frame clipping outline.
DDIF$K_FMT_FLUSH_PATH_BOTH	The text string is imaged such that the left alignment point of the first character is aligned with the start of the text path, and the right alignment point of the last character is aligned with the end of the path. If the string layout path is shorter than the text string in this case, the text will be forced onto the path by reducing the escapement of the characters in the string.

The default is DDIF$K_FLUSH_PATH_BEGIN.

- A layout path orientation indicator (type DDIF$_SGA_LAYPTH_ORIENTATION_C) that selects the format used to specify the orientation of characters along the path. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_PATH_FIXED	The characters are oriented at a fixed angle relative to the current frame. In this case, the DDIF$_SGA_LAYPTH_ORIENTATION item is encoded as a single-precision floating-point value.
DDIF$K_PATH_RELATIVE	The characters are oriented at an angle that is relative to the slope of the path at the point at which the character is imaged. In this case, the DDIF$_SGA_LAYPTH_ORIENTATION item is encoded as an enumeration. Valid values are as follows:

DDIF$K_RIGHT_ANGLE_RIGHT	An angle at 0 degrees with respect to the current coordinate system.
DDIF$K_RIGHT_ANGLE_LEFT	An angle at 180 degrees with respect to the current coordinate system.
DDIF$K_RIGHT_ANGLE_UP	An angle at 90 degrees with respect to the current coordinate system.
DDIF$K_RIGHT_ANGLE_DOWN	An angle at 270 degrees with respect to the current coordinate system.

- A layout path orientation item (type DDIF$_SGA_LAYPTH_ORIENTATION) that specifies the actual character orientation along the path. This item is encoded as a variable.
- A horizontal alignment item (type DDIF$_SGA_LAYPTH_H_ALIGN) that specifies the horizontal alignment point for characters along a path. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_PATH_NORMAL_HORIZONTAL	Characters are horizontally aligned relative to the active position using the value defined for normal horizontal alignment in Table 6–27.
DDIF$K_PATH_LEFTLINE	Characters are horizontally aligned such that the active position is a point on the left line of the character.
DDIF$K_PATH_CENTERLINE	Characters are horizontally aligned such that the active position is a point on the center line of the character.
DDIF$K_PATH_RIGHTLINE	Characters are horizontally aligned such that the active position is a point on the right line of the character.

  The default is DDIF$K_PATH_NORMAL_HORIZONTAL.
- A vertical alignment item (type DDIF$_SGA_LAYPTH_V_ALIGN) that specifies the vertical alignment point for characters along a path. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_PATH_NORMAL_VERTICAL	The character is aligned using the value defined for normal vertical alignment in Table 6–27.

DDIF$K_PATH_BASELINE	Characters are vertically aligned such that the active position is a point on the baseline.
DDIF$K_PATH_CAPLINE	Characters are vertically aligned such that the active position is a point on the cap line.
DDIF$K_PATH_BOTTOMLINE	Characters are vertically aligned such that the active position is a point on the bottom line.
DDIF$K_PATH_HALFLINE	Characters are vertically aligned such that the active position is a point on the half line.
DDIF$K_PATH_TOPLINE	Characters are vertically aligned such that the active position is a point on the top line.

The default is DDIF$K_PATH_NORMAL_VERTICAL.

Table 6–27 lists the normal alignments for the various orientations.

Table 6–27 Normal Alignment

Orientation	Up	Right	Down	Left	Angle
Horizontal	LEFTLINE	CENTERLINE	RIGHTLINE	CENTERLINE	CENTERLINE
Vertical	BASELINE	BOTTOMLINE	BASELINE	TOPLINE	HALFLINE

6.20.8.3 Position-Relative Layout

Position-relative layout specifies that the characters in the segment are positioned relative to the current text position. This layout type is selected by specifying DDIF$_SGA_LAYOUT_C as DDIF$K_RELATIVE_LAYOUT. The layout itself is specified using the following items:

- A horizontal ratio numerator item (type DDIF$_SGA_LAYREL_H_RATIO_N) that specifies the magnitude of the escapement ratio to be used in determining the horizontal position of the character relative to the current text. This item is encoded as an integer.
- A horizontal ratio denominator item (type DDIF$_SGA_LAYREL_H_RATIO_D) that specifies the units of precision used in the escapement ratio to be used in determining the horizontal position of the character relative to the current text. This item is encoded as an integer.
- A relative horizontal position constant indicator (type DDIF$_SGA_LAYREL_H_CONSTANT_C) that indicates whether the horizontal position is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A relative horizontal position constant item (type DDIF$_SGA_LAYREL_H_CONSTANT) that specifies a constant measurement to be used as an escapement. This item is encoded as a variable.

- A vertical ratio numerator item (type DDIF$_SGA_LAYREL_V_RATIO_N) that specifies the magnitude of the escapement ratio to be used in determining the vertical position of the character relative to the current text. This item is encoded as an integer.
- A vertical ratio denominator item (type DDIF$_SGA_LAYREL_V_RATIO_D) that specifies the units of precision used in the escapement ratio to be used in determining the vertical position of the character relative to the current text. This item is encoded as an integer.
- A relative vertical position constant indicator (type DDIF$_SGA_LAYREL_V_CONSTANT_C) that indicates whether the vertical position is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A relative vertical position constant item (type DDIF$_SGA_LAYREL_V_CONSTANT) that specifies a constant measurement to be used as an escapement. This item is encoded as a variable.

6.20.8.4 Text Position

The text position layout (selected by specifying DDIF$_SGA_LAYOUT_C as DDIF$K_POSITION_LAYOUT) is specified using a text position indicator (type DDIF$_SGA_LAYPOS_TEXT_POSITION) that indicates the relational position of the segment relative to the current baseline. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_TEXT_POS_BASE	The text in the segment forms the base for special positions in the segment. The text rests on the current baseline.
DDIF$K_TEXT_POS_L_SUBSCRIPT	The right alignment position of the last character of the subscript string is placed at the left subscript position.
DDIF$K_TEXT_POS_L_SUPERSCRIPT	The right alignment position of the last character of the superscript string is placed at the left superscript position.
DDIF$K_TEXT_POS_R_SUBSCRIPT	The left alignment position of the first character of the subscript string is placed at the right subscript position.
DDIF$K_TEXT_POS_R_SUPERSCRIPT	The left alignment position of the first character of the superscript string is placed at the right superscript position.
DDIF$K_TEXT_POS_TOP_CENTER	The segment is centered above the total string of the base.
DDIF$K_TEXT_POS_BOTTOM_CENTER	The segment is centered below the total string of the base.
DDIF$K_TEXT_POS_RUBI	The segment is centered above the total string of the base.

There are certain restrictions that must be observed when specifying text position.

- No changes in segment layout can take place within positional layout segments unless those segments are in a frame.

- Frames in positional layout segments must have an inline frame position.
- The base segment must be the first child of the parent segment.

6.20.9 Font Definitions

The font definitions item (type DDIF$_SGA_FONT_DEFNS) specifies a list of fonts defined for use within the segment. This item is encoded as a sequence of DDIF$_FTD aggregates. (For more information on the DDIF$_FTD aggregate, see Section 6.22.) Each font definition assigns a number to a font by which it is referenced within the segment. Initially, there are no font definitions.

6.20.10 Pattern Definitions

The pattern definitions item (type DDIF$_SGA_PATTERN_DEFNS) specifies a list of patterns and solid colors defined for use within the segment. This item is encoded as a sequence of DDIF$_PTD aggregates. For more information on the DDIF$_PTD aggregate, see Section 6.25.

6.20.11 Path Definitions

The path definitions item (type DDIF$_SGA_PATH_DEFNS) specifies a list of predefined paths that can be referenced within the segment. This item is encoded as a sequence of DDIF$_PHD aggregates. (For more information on the DDIF$_PHD aggregate, see Section 6.24.) Initially, no paths are defined.

6.20.12 Line-Style Definitions

The line-style definitions item (type DDIF$_SGA_LINE_STYLE_DEFNS) specifies a list of predefined line styles that can be referenced within the document. This item is encoded as a sequence of DDIF$_LSD aggregates. For more information on the DDIF$_LSD aggregate, see Section 6.23.

6.20.13 Content Definitions

The optional content definitions item (type DDIF$_SGA_CONTENT_DEFNS) specifies a list of content definitions that can be referenced within the segment. This item is encoded as a sequence of DDIF$_CTD aggregates. (For more information on the DDIF$_CTD aggregate, see Section 6.21.) Initially, there are no content definitions.

6.20.14 Type Definitions

The type definitions item (type DDIF$_SGA_TYPE_DEFNS) specifies a list of segment type definitions that can be referenced within the segment. This item is encoded as a sequence of DDIF$_TYD aggregates. (For more information on the DDIF$_TYD aggregate, see Section 6.27.) Initially, there are no type definitions.

6.20.15 Text Attributes

The text attribute items define the default presentation attributes of text within the segment. The text attribute items fall into the following groups:

- Text mask pattern
- Text font
- Text rendition
- Text size
- Text direction
- Text character decimal alignment
- Text leader attributes
- Text kerning
- Text kerning delta attributes
- Text letter spacing

The items in each of these groups are discussed in the following sections.

6.20.15.1 Text Mask Pattern

The text mask pattern item (type DDIF$_SGA_TXT_MASK_PATTERN) specifies the pattern and color of glyphs, using an index into the current list of patterns. This item is encoded as an integer. In addition to user-defined pattern numbers, several predefined patterns are supplied. These patterns are listed in Appendix F.

The text mask pattern is initialized to DDIF$K_PATT_FOREGROUND, which corresponds to DDIF fill pattern number 2.

6.20.15.2 Text Font

The text font item (type DDIF$_SGA_TXT_FONT) specifies the font in which the text is rendered. This item is encoded as an integer. The text font is specified as an index into the list of fonts defined by the current segment and/or parent segment. The character set specified in the font identifier of the referenced font definition must match the character set of the text content that appears in the segment. The text font is initialized to font number 1.

6.20.15.3 Text Rendition

The text rendition item (type DDIF$_SGA_TXT_RENDITION) specifies one or more text renditions. (A text rendition modifies the appearance of characters or strings.) This item is encoded as an array of type enumeration. Valid values are as follows:

DDIF$K_RND_DEFAULT	The text is imaged as defined by the current "nonrendition" text presentation attributes, without any additional change in rendition.
DDIF$K_RND_HIGHLIGHT	The text is rendered in a higher than normal intensity, or a heavier typeface. This rendition is usually used when the document is intended for a video display device.
DDIF$K_RND_FAINT	The text is rendered in a lower than normal intensity. This rendition is usually used when the document is intended for a video display device.
DDIF$K_RND_ITALIC	The text is rendered in the italic or slant style of the current font.
DDIF$K_RND_NORMAL	The text is rendered in normal intensity.
DDIF$K_RND_SLOW_BLINK	The intensity of the characters alternates between two states at a relatively slow rate. This is used only for documents intended primarily for video display. The fallback rendition on static displays is text in a different color.
DDIF$K_RND_FAST_BLINK	The intensity of the characters alternates between two states at a relatively high rate. This is used only for documents intended primarily for video display. The fallback rendition on static displays is text in a different color.
DDIF$K_RND_NO_BLINK	The intensity of the characters is steady.
DDIF$K_RND_NEGATIVE	The normal relationship between the text foreground and background color is reversed.
DDIF$K_RND_POSITIVE	The text color is not reversed.
DDIF$K_RND_CONCEAL	The text string occupies the same space as usual but the characters are not imaged. Note that underlines, overlines, and cross-outs are not concealed by this attribute.
DDIF$K_RND_NO_CONCEAL	The text is imaged rather than concealed.
DDIF$K_RND_UNDERLINE	A line parallel with the text path is drawn under the text. Note that spaces are underlined except when the space is omitted from the presentation form by word wrap and justification software.
DDIF$K_RND_2_UNDERLINE	The text is underlined twice, with an implementation-defined distance between the lines.
DDIF$K_RND_NO_UNDERLINE	Text is not underlined.

DDIF$K_RND_CROSS_OUT	A line that is thin compared to the weight of the text is drawn through the string. The location of the line is determined by the implementation.
DDIF$K_RND_BOX	The text is enclosed in a box. The size of the box is the smallest that will enclose the text without touching any character.
DDIF$K_RND_ENCIRCLE	The text is enclosed in an ellipse or rounded rectangle. The total area of the ellipse is the minimum that will enclose the text without touching any character.
DDIF$K_RND_OVERLINE	A line is drawn parallel to the text path and above it relative to the text.
DDIF$K_RND_IDEO_UNDERLINE	A line parallel to the text path is drawn under the text, or along the right side of text that is presented vertically.
DDIF$K_RND_IDEO_2_UNDERLINE	Two lines parallel to the text path are drawn under the text, or along the right side of text that is presented vertically.
DDIF$K_RND_IDEO_OVERLINE	A line parallel to the text path is drawn over the text, or along the left side of text that is presented vertically.
DDIF$K_RND_IDEO_2_OVERLINE	Two lines parallel to the text path are drawn under the text, or along the left side of text that is presented vertically.
DDIF$K_RND_IDEO_STRESS	Characters have ideographic stress markers.

The initial value of this item is DDIF$K_RND_DEFAULT.

6.20.15.4 Text Size

The text size attributes specify the height and width of the text in the segment. These attributes are specified using the following items:

- A text height indicator (type DDIF$_SGA_TXT_HEIGHT_C) that indicates whether the text height is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A text height item (type DDIF$_SGA_TXT_HEIGHT) that specifies the height of the text in the segment. This item is encoded as a variable. The current font of the segment is scaled if the type size specified in its font metrics definition does not equal the text size. The initial value of this item is 1.
- A text size numerator item (type DDIF$_SGA_TXT_SET_SIZE_N) that specifies the magnitude of the ratio of the actual character width to the design width for the current font at the current text height. This item is encoded as an integer with a default value of 1.
- A text size denominator item (type DDIF$_SGA_TXT_SET_SIZE_D) that specifies the units of precision used in the character width ratio. This item is encoded as an integer with a default value of 100.

6.20.15.5 Text Direction

The text direction item (type DDIF$_SGA_TXT_DIRECTION) defines the placement of characters along the current text path with respect to the logical ordering of the characters. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_TXT_DIR_FORWARD	The text proceeds in the direction of the path.
DDIF$K_TXT_DIR_BACKWARD	The text proceeds opposite the direction of the path.

The initial value of this item is DDIF$K_TXT_DIR_FORWARD.

6.20.15.6 Text Character Decimal Alignment

The text character decimal alignment item (type DDIF$_SGA_TXT_DEC_ALIGNMENT) specifies the characters in a decimal-aligned tab field on which the alignment occurs. This item is encoded as an array of type character string. The order in which the characters are listed indicates their alignment priority. The initial value of this item contains the following characters:

Period	.
Comma	,
Close parenthesis	)

6.20.15.7 Text Leader Attributes

The optional text leader attributes items describe the presentation attributes of leaders. Leaders are rows of dashes or dots that are used to guide the eye across the page. The text leader attributes are controlled using the following items:

- An optional leader space indicator (type DDIF$_SGA_TXT_LEADER_SPACE_C) that indicates whether the leader space is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An optional leader space item (type DDIF$_SGA_TXT_LEADER_SPACE) that specifies the amount of additional space that is inserted between leader characters. This item is encoded as a variable. The initial value of this item is 0.
- An optional leader bullet item (type DDIF$_SGA_TXT_LEADER_BULLET) that specifies the text string, usually a single character, that is used to fill leader space. This item is encoded as a character string. Characters are selected from the current font.
- An optional leader alignment item (type DDIF$_SGA_TXT_LEADER_ALIGN) that specifies the alignment of leaders. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_ALIGNED_LEADER	Leader characters should be aligned.
DDIF$K_STAGGERED_LEADER	The center points of leader characters should alternate.
DDIF$K_NON_ALIGNED_LEADER	No alignment has been selected.

- An optional leader style item (type DDIF$_SGA_TXT_LEADER_STYLE) that specifies the type of leader to use. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_X_RULE_LEADER	Draws a horizontal rule.
DDIF$K_BULLET_LEADER	Uses the current leader-bullet string.

6.20.15.8 Text Kerning

In typesetting, **kerning** is defined as the operation of subtracting the space between two characters so that they appear closer together. This concept is used in proportionally spaced fonts to make the distance between characters appear equal. The text pair kerning item (type DDIF$_SGA_TXT_PAIR_KERNING) specifies a Boolean value that controls whether text in the segment is kerned based on kerning pair tables for the current font. This item is encoded as a Boolean value. If no kerning pair information is available for the font, all kerning deltas for that font are assumed to be zero. The initial value for this item is false.

6.20.16 Line Attributes

The line attributes are specified using the following items:

- An optional line width indicator (type DDIF$_SGA_LIN_WIDTH_C) that indicates whether the line width is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An optional line width item (type DDIF$_SGA_LIN_WIDTH) that specifies the width of the line in Basic Measurement Units (BMUs). This item is encoded as a variable. The initial value is 0, indicating the thinnest visible line width on the display device.
- An optional line style item (type DDIF$_SGA_LIN_STYLE) that specifies the pattern used for drawing lines as either a standard representation or as a pattern to replicate. This item is encoded as an integer. Valid values are listed in Table 6–28.

Table 6–28 Line Style

Line Style	Repeating Pattern
DDIF$K_SOLID_LINE_STYLE	1111
DDIF$K_DASH_LINE_STYLE	1100
DDIF$K_DOT_LINE_STYLE	1010
DDIF$K_DASH_DOT_LINE_STYLE	11010

The initial line style is DDIF$K_SOLID_LINE_STYLE.

- An optional line pattern size indicator (type DDIF$_SGA_LIN_PATTERN_SIZE_C) that indicates whether the pattern size is specified as a variable or constant value. This item is encoded as a measurement enumeration.

- An optional line pattern size item (type DDIF$_SGA_LIN_PATTERN_SIZE) that specifies the size of the line pattern. This item is encoded as a variable. The initial value of this item is 0. This item acts as a multiplier for the line pattern specified by DDIF$_LSD_PATTERN.
- An optional line mask pattern (type DDIF$_SGA_LIN_MASK_PATTERN) that specifies the mask pattern of the line as an index into the current pattern definitions. This item is encoded as an integer. In addition to the user-defined pattern numbers, several predefined patterns are provided. These patterns are illustrated in Appendix F.

  The initial line mask pattern is DDIF$K_PATT_FOREGROUND, which corresponds to pattern number 2.
- An optional line-end start item (type DDIF$_SGA_LIN_END_START) that determines the shape of the line ending at the first point on the path that describes the line. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_BUTT_LINE_END	The line begins exactly at the starting point, with a flat end.
DDIF$K_ROUND_LINE_END	The line begins with a circle the width of the line centered at the starting point.
DDIF$K_SQUARE_LINE_END	The line begins with a square the width of the line centered at the starting point.
DDIF$K_ARROW_LINE_END	The line begins with a triangular area, with the same mask pattern as the line itself, whose base is three times the width of the line and centered on the starting point of the line. The apex of the triangle is on a line tangent to the direction of the line at its starting point. The distance from the apex to the beginning of the line is equal to the width of the line.

  The initial value of this item is DDIF$K_ROUND_LINE_END.

Note: The DDIF$_SGA_LIN_END_START and DDIF$_LIN_END_FINISH items are only different for lines that have an arrow line ending in cases where one end has an arrow and the other does not.

- An optional line-end finish item (type DDIF$_SGA_LINE_END_FINISH) that determines the shape of the line ending. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_BUTT_LINE_END	The line ends exactly at the end point, with a flat end.
DDIF$K_ROUND_LINE_END	The line ends with a circle the width of the line centered at the end point.
DDIF$K_SQUARE_LINE_END	The line ends with a square the width of the line centered at the end point.

DDIF$K_ARROW_LINE_END	The line ends with a triangular area, with the same mask pattern as the line itself, whose base is three times the width of the line and centered on the end point of the line. The apex of the triangle is on a line tangent to the direction of the line at its end point. The distance from the apex to the line end is equal to the width of the line.

The initial value of this item is DDIF$K_ROUND_LINE_END.

- An optional line-end size indicator (type DDIF$_SGA_LIN_END_SIZE_C) that indicates whether the ending size of the line is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An optional line-end size item (type DDIF$_SGA_LIN_END_SIZE) that specifies the ending size of the line. This item is encoded as a variable. The initial value of this item is 0.
- An optional line join item (type DDIF$_SGA_LIN_JOIN) that specifies an integer with defined values that determine the shape of line joins. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_MITERED_LINE_JOIN	The join of the line is mitered.
DDIF$K_ROUNDED_LINE_JOIN	The join of the line is rounded.
DDIF$K_BEVELED_LINE_JOIN	The join of the line is beveled.

The initial value of this item is DDIF$K_ROUNDED_LINE_JOIN.

- An optional miter limit numerator item (type DDIF$_SGA_LIN_MITER_LIMIT_N) that specifies the magnitude of the allowed ratio between the length of the mitered line joint and the width of the line. This item is encoded as an integer. When the miter limit is exceeded, the joint is beveled instead. The initial value for this item is 10.
- An optional miter limit denominator item (type DDIF$_SGA_LIN_MITER_LIMIT_D) that specifies the units of precision of the allowed ratio between the length of the mitered line joint and the width of the line. This item is encoded as an integer. The initial value for this item is 100.
- The line interior pattern item (type DDIF$_SGA_LIN_INTERIOR_PATTERN) specifies the fill pattern or solid color to be used for objects designated as filled or as having a background, including polylines, arcs, curves, fill area sets, frame borders, and galley borders. This item is encoded as an integer. In addition to the user-defined pattern numbers, several predefined patterns are provided. These patterns are described in Appendix F.

  The initial value for this item is DDIF$K_PATT_BACKGROUND. The application of the fill pattern is controlled by a flag on the object to be filled.

6.20.17 Marker Attributes

The marker attributes specify the default presentation attributes for markers within the segment. The marker attributes are specified using the following items:

- An optional marker style item (type DDIF$_SGA_MKR_STYLE) that specifies the symbol used as the marker. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_DOT_MARKER	Dot marker
DDIF$K_PLUS_MARKER	Plus sign marker
DDIF$K_ASTERISK_MARKER	Asterisk marker
DDIF$K_CIRCLE_MARKER	Circle marker
DDIF$K_CROSS_MARKER	Diagonal cross marker

  The marker type is initially defined to be DDIF$K_DOT_MARKER.

- An optional marker mask pattern item (type DDIF$_SGA_MKR_MASK_PATTERN) that defines an index into the pattern list for markers. This item is encoded as an integer. In addition to the user-defined pattern numbers, several predefined patterns are provided. These patterns are described in Appendix F.

  The initial marker mask pattern is DDIF$K_PATT_FOREGROUND.

- An optional marker size indicator (type DDIF$_SGA_MKR_SIZE_C) that indicates whether the marker size is specified as a variable or constant value. This item is encoded as a measurement enumeration.

- An optional marker size item (type DDIF$_SGA_MKR_SIZE) that defines the size of markers in BMUs (which can be scaled). This item is encoded as a variable. The initial marker size is the smallest marker size supported by the application, indicated by a value of 0.

6.20.18 Galley Attributes

Galley attributes apply to galleys defined within a segment. The galley attributes of a segment containing text within the document body do not affect the layout of text. Thus, galley attributes are normally used only in the context of defining galleys in a page frame or in a floating frame that has galleys.

The galley attributes item (type DDIF$_SGA_GLY_ATTRIBUTES) controls the presentation attributes of galleys in a segment. This item is encoded as the handle of a DDIF$_GLA aggregate. For more information on the DDIF$_GLA aggregate, see Section 6.37.

6.20.19 Image Attributes

The image attributes control the default presentation attributes of images within the segment. The image attributes are specified using the following items:

- An optional private data item (type DDIF$_SGA_IMG_PRIVATE_DATA) that allows for the inclusion of application-private data needed for the presentation of image data. This item is encoded as a sequence of DDIF$_PVT aggregates. For more information on the DDIF$_PVT aggregate, see Section 6.15.2.
- An optional pixel path item (type DDIF$_SGA_IMG_PIXEL_PATH) that specifies the direction of the pixel capture path along an individual scanline. This item is encoded as an integer value that corresponds to an angular measure in minutes of an arc with respect to the standard orientation of an image. To ensure compatibility with ISO and CCITT standards, values equivalent to 0, 90, 180, and 270 degrees should be used. The default is 0 degrees.
- An optional line progression item (type DDIF$_SGA_IMG_LINE_PROGRESSION) that specifies the direction of scanline capture across the image plane. This item is encoded as an integer value that corresponds to an angular measure in minutes of an arc with respect to the standard orientation of an image. To ensure compatibility with ISO and CCITT standards, values equivalent to 90 and 270 degrees should be used. The initial value is 16200, which is equivalent to 270 degrees expressed in minutes.
- An optional pixel path aspect ratio item (type DDIF$_SGA_IMG_PP_PIXEL_DIST) that specifies the ratio of the distance between pixel centers along the pixel path and along the line progression path. This item is encoded as an integer. The default ratio is 1:1 or 1.
- An optional line progression path aspect ratio item (type DDIF$_SGA_IMG_LP_PIXEL_DIST) that specifies the aspect ratio along the line progression path. This item is encoded as an integer. The initial ratio is 1:1 or 1.
- A brightness polarity item (type DDIF$_SGA_IMG_BRT_POLARITY) that is used to interpret the manner in which pixel values represent minimum and maximum intensity; that is, whether a value of 0 represents the minimum or maximum intensity value. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_ZERO_MAX_INTENSITY	Zero represents the maximum intensity.
DDIF$K_ZERO_MIN_INTENSITY	Zero represents the minimum intensity.

  The default is DDIF$K_ZERO_MAX_INTENSITY.
- An optional grid type item (type DDIF$_SGA_IMG_GRID_TYPE) that identifies the physical format of the pixel grid. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_RECTANGULAR_GRID	Rectangular grid

DDIF$K_HEX_EVEN_INDENT	Hexagonal grid with even indentation
DDIF$K_HEX_ODD_INDENT	Hexagonal grid with odd indentation

The initial value is DDIF$K_RECTANGULAR_GRID.

- An optional timing descriptor item (type DDIF$_SGA_IMG_TIMING_DESC) that signifies that the frame containing multiple image data descriptors (or multiple image content elements) is a motion sequence. In a motion sequence, each image content element represents a single picture cell or cell in the sequence. This item is encoded as a binary relative time. This value is initially absent.
- An optional spectral component mapping item (type DDIF$_SGA_IMG_SPECTRAL_MAPPING) that designates the correlation between the physical image data and the spectral components of an image. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_PRIVATE_MAP	Correlation is privately mapped.
DDIF$K_MONOCHROME_MAP	Correlation is monochrome mapped.
DDIF$K_GENERAL_MAP	Correlation is general multispectral.
DDIF$K_LUT_MAP	Correlation is lookup table mapped.
DDIF$K_RGB_MAP	Correlation is RGB (red/green/blue) mapped.
DDIF$K_CMY_MAP	Correlation is CMY (cyan/magenta/yellow) mapped.
DDIF$K_YUV_MAP	Correlation is YUV mapped.
DDIF$K_HSV_MAP	Correlation is HSV (hue saturation value) mapped.
DDIF$K_HIS_MAP	Correlation is HIS (hue intensity saturation) mapped.
DDIF$K_YIQ_MAP	Correlation is YIQ mapped.

The initial value of this item is DDIF$K_MONOCHROME_MAP.

- An optional lookup table indicator (type DDIF$_SGA_IMG_LOOKUP_TABLES_C) that specifies the type of lookup table to be specified. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_PRIVATE_LUT	The lookup table contains a sequence of one or more named values, where each named value contains lookup table information that is private to the creator of the document. In this case, DDIF$_SGA_IMG_LOOKUP_TABLES is encoded as a sequence of DDIF$_PVT aggregates, described in Section 6.15.2.
DDIF$K_RGB_LUT	The lookup table contains a sequence of lookup table entries, where each entry describes a lookup table index corresponding to the pixel that it maps, and describes the red, gree, and blue intensities that are generated for that pixel. The index corresponds to the integer value of the lookup-table-mapped pixel, and can range in value between 0 and $2^{16} - 1$. In this case, DDIF$_SGA_IMG_LOOKUP_TABLES is encoded as a sequence of DDIF$_RGB aggregates, described in Section 6.31.

- An optional lookup table item (type DDIF$_SGA_IMG_LOOKUP_TABLES) that contains an octet string containing application private lookup tables. This item is encoded as a variable.
- An optional component wavelength indicator (type DDIF$_SGA_IMG_COMP_WAVELENGTH_C) that specifies the wavelength being supplied by the DDIF$_SGA_IMG_COMP_WAVELENGTH item. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_APPLICATION_WAVELENGTH	Specifies application-specific data for each component. In this case, the DDIF$_SGA_IMG_COMP_WAVELENGTH item must be encoded as an array of type string.
DDIF$K_WAVELENGTH_MEASURE	Specifies a wavelength measure in angstroms that can represent either a single wavelength or the most significant frequency within a range of frequencies. In this case, the DDIF$_SGA_IMG_COMP_WAVELENGTH item must be encoded as an array of type integer.
DDIF$K_WAVELENGTH_BAND_ID	Specifies the spectral band identification codes that are permitted by the application. In this case, the DDIF$_SGA_IMT_COMP_WAVELENGTH item must be encoded as an array of type string.

- An optional component wavelength information item (type DDIF$_SGA_IMG_COMP_WAVELENGTH) that specifies the information selected by DDIF$_SGA_IMG_COMP_WAVELENGTH_C. This item is encoded as a variable.

6.20.20 Image Component Space Attributes

The image component space attributes describe characteristics of the component space. These attributes are specified using the following items:

- A component space organization item (type DDIF$_SGA_IMG_COMP_SPACE_ORG) that designates how the component space data is physically organized. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_FULL_COMPACTION	Indicates that all the component bits for a pixel are collected into a single data plane and are adjacent to one another within the physical bit field designated as a single logical pixel. For example, in a 3-3-2 RGB image, a single pixel comprises three bits of red, followed by three bits of green, followed by two bits of blue. The next logical pixel is of identical composition. Aside from possible padding at the end of the component bits for each pixel, this organization implies maximal adjacency between uncompressed pixel component data. This organization always implies that only one data plane exists for each content element.
DDIF$K_PARTIAL_EXPANSION	Indicates that the component bits for a pixel are spread across multiple data planes in the following manner: the pixel data for each component occupies a separate data plane. This organization only applies to multispectral images. For example, the data for an RGB image can be partitioned such that the first plane contains the red bits for all pixels, the second plane the green bits, and the third plane the blue bits, for a total of three planes.
DDIF$K_FULL_EXPANSION	Indicates that the component bits for a pixel are spread across multiple data planes in the following manner: each bit per component exists in a separate data plane, so that the logical index into the pixel data of a single plane physically references a bit field that is a single bit in length, and the logical index into the data plane set references the pixel component bits by order of significance. For example, the data for a 3-3-2 RGB image would occupy eight data planes: three for red, three for green, and two for blue. In this organization, the pixel bits of a gray-scale image could be expanded by significance into separate data planes.

The initial value of this item is DDIF$K_FULL_EXPANSION.

- An optional data-planes-per-pixel item (type DDIF$_SGA_IMG_PLANES_PER_PIXEL) that specifies the number of data planes per pixel (and consequently per image) used to span the component space. This item is encoded as an integer whose value corresponds to the number of image data units used to represent a particular image. The initial value is 1.
- An optional data plane significance item (type DDIF$_SGA_IMG_PLANE_SIGNIF) that only has meaning for image data organized in Expanded Component Sequential Form. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_LSB_MSB	Least significant bit to most significant bit.
DDIF$K_MSB_LSB	Most significant bit to least significant bit.

The default is DDIF$K_LSB_MSB.

- An optional number-of-components item (type DDIF$_SGA_IMG_NUMBER_OF_COMP) that specifies the number of spectral components in a multispectral image. This item is encoded as an integer.
- An optional bits-per-component item (type DDIF$_SGA_IMG_BITS_PER_COMP) that specifies the number of bits used for each image component in a data plane. The sum of all bits per component for all data planes equals the number of bits per pixel. This item is encoded as an array of type integer.

6.20.21 Frame Parameters

The frame parameters cause the content of the segment to be bounded within a frame whose origin is located at the lower left-hand corner of the frame. The frame parameters fall into the following categories:

- Frame flags
- Frame bounding box
- Frame outline
- Frame clipping
- Frame position
- Frame content transformation
- Frame border attributes
- Frame background color
- Frame galleys

The items used to specify each of these categories are discussed in the following sections. Note that there are no initial frame parameters.

6.20.21.1 Frame Flags

The optional frame flags item (type DDIF$_SGA_FRM_FLAGS) specifies the flags that control the presentation of the frame and/or text around the frame. This item is encoded as a longword. Valid frame flag values are as follows:

DDIF$M_FLOW_AROUND	Document text flows around the path given by the frame outline.
DDIF$M_FRAME_BORDER	A line is drawn around the frame outline using the current line attributes.

DDIF$M_FRAME_BACKGROUND_FILL	The frame is filled with the pattern or color given by the current line interior fill item (DDIF$_SGA_LIN_INTERIOR_PATTERN) before the content of the frame is imaged.

6.20.21.2 Frame Bounding Box

The frame bounding box items specify a rectangular area that outlines the frame and defines the origin of the frame. The frame bounding box is described using the following items:

- A lower left corner *x* position indicator (type DDIF$_SGA_FRM_BOX_LL_X_C) that indicates whether the lower left corner *x*-coordinate is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A lower left corner *x* position item (type DDIF$_SGA_FRM_BOX_LL_X) that specifies the *x*-coordinate of the lower left corner of the frame bounding box. This item is encoded as a variable.
- A lower left corner *y* position indicator (type DDIF$_SGA_FRM_BOX_LL_Y_C) that indicates whether the lower left corner *y*-coordinate is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A lower left corner *y* position item (type DDIF$_SGA_FRM_BOX_LL_Y) that specifies the *y*-coordinate of the lower left corner of the frame bounding box. This item is encoded as a variable.
- An upper right corner *x* position indicator (type DDIF$_SGA_FRM_BOX_UR_X_C) that indicates whether the upper right corner *x*-coordinate is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An upper right corner *x* position item (type DDIF$_SGA_FRM_BOX_UR_X) that specifies the *x*-coordinate of the upper right corner of the frame bounding box. This item is encoded as a variable.
- An upper right corner *y* position indicator (type DDIF$_SGA_FRM_BOX_UR_Y_C) that indicates whether the upper right corner *y*-coordinate is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An upper right corner *y* position item (type DDIF$_SGA_FRM_BOX_UR_Y) that specifies the *y*-coordinate of the upper right corner of the frame bounding box. This item is encoded as a variable.

6.20.21.3 Frame Outline

The optional frame outline item (type DDIF$_SGA_FRM_OUTLINE) specifies the path to which text flowing around the frame is aligned. This item is encoded as a sequence of DDIF$_PTH aggregates. For more information on the DDIF$_PTH aggregate, see Section 6.19.

If the frame outline item is not specified, the default path is the path given by the bounding box. The path defined by the frame outline is constrained to fit within the specified bounding box.

6.20.21.4 Frame Clipping

The optional frame clipping item (type DDIF$_SGA_FRM_CLIPPING) specifies the clipping path of the frame, specified as a path whose coordinates are relative to the origin (0,0) of the frame. This item is encoded as a sequence of DDIF$_PTH aggregates. For more information on the DDIF$_PTH aggregate, see Section 6.19.

The path that is specified as the clipping region is constrained to fit within the specified bounding box, and it can be different from the outline. No content is imaged outside the clipping region. The inside of the clipping region is determined by the odd winding rule. (The odd winding rule states that, if a ray is drawn from a point to infinity, the origin of the ray is considered inside the area (and hence will be filled) if it crosses the area border an odd number of times.)

6.20.21.5 Frame Position

The frame position items specify the fixed or preferred position of the frame relative to the enclosing frame. The frame position information is described by first selecting the type of position, and then specifying the appropriate information for that position type. (The origin of a frame is located at the lower lefthand corner.) To select the position type, a position item (type DDIF$_SGA_FRM_POSITION_C) is used. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_FRAME_FIXED	The origin of the frame is placed at the specified position relative to the current frame of reference (a page or a frame). If you specify this position type, you must supply values for the items DDIF$_SGA_FRMFXD_POSITION_X_C through DDIF$_SGA_FRMFXD_POSITION_Y.
DDIF$K_FRAME_INLINE	The origin of the frame is positioned along the current text path. The frame behaves like a character the width of the frame. If you specify this position type, you must supply values for the items DDIF$_SGA_FRMINL_BASE_OFFSET_C and DDIF$_SGA_FRMINL_BASE_OFFSET.
DDIF$K_FRAME_GALLEY	The origin of the frame is placed at a preferred position within the current galley. This type of frame positioning should be specified only for content using galley-based layout. If you specify this position type, you must supply values for the items DDIF$_SGA_FRMGLY_VERTICAL and DDIF$_SGA_FRMGLY_HORIZONTAL.
DDIF$K_FRAME_MARGIN	The origin of the frame is placed at a preferred position relative to the current position, but outside the current galley. This type of frame positioning should be specified only for content using galley-based layout. If you specify this position type, you must supply values for the items DDIF$_SGA_FRMMAR_BASE_OFFSET_C through DDIF$_SGA_FRMMAR_HORIZONTAL.

The following sections discuss each of these frame positions.

6.20.21.5.1 Fixed Frame Parameters

The fixed position frame parameters (selected by specifying DDIF$_SGA_FRM_POSITION_C as DDIF$K_FRAME_FIXED) are specified using the following items:

- An *x* position indicator (type DDIF$_SGA_FRMFXD_POSITION_X_C) that indicates whether the *x* position is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An *x* position item (type DDIF$_SGA_FRMFXD_POSITION_X) that specifies the *x* position of the origin of the frame. This item is encoded as a variable.
- A *y* position indicator (type DDIF$_SGA_FRMFXD_POSITION_Y_C) that indicates whether the *y* position is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A *y* position item (type DDIF$_SGA_FRMFXD_POSITION_Y) that specifies the *y* position of the origin of the frame. This item is encoded as a variable.

6.20.21.5.2 Inline Frame Parameters

The inline position frame parameters (selected by specifying DDIF$_SGA_FRM_POSITION_C as DDIF$K_FRAME_INLINE) are specified using the following items:

- A base offset indicator (type DDIF$_SGA_FRMINL_BASE_OFFSET_C) that indicates whether the base offset value is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A base offset item (type DDIF$_SGA_FRMINL_BASE_OFFSET) that specifies the vertical offset of the origin (0,0) of the frame relative to the baseline on which the frame is positioned. This item is encoded as a variable.

6.20.21.5.3 Galley Frame Parameters

The galley frame parameters (selected by specifying DDIF$_SGA_FRM_POSITION_C as DDIF$K_FRAME_GALLEY) are specified using the following items:

- A vertical galley frame parameter (type DDIF$_SGA_FRMGLY_VERTICAL) that defines a standard or private label that specifies the preferred vertical positioning of the lower edge of the frame. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_FRMGLY_BELOW_CURRENT	The frame is positioned so that the top of the frame is on what would be the next baseline.
DDIF$K_FRMGLY_BOTTOM	The frame is positioned so that the lower edge of the frame is on the lower edge of the galley in which it is imaged.

DDIF$K_FRMGLY_TOP	The frame is positioned so that the upper edge of the frame is on the upper edge of the galley in which it is imaged.

- A horizontal galley frame parameter (type DDIF$_SGA_FRMGLY_HORIZONTAL) that specifies the horizontal position of the frame relative to its reference frame. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_FMT_FLUSH_PATH_BEGIN	The first character is imaged at the start of the text path, and successive characters are imaged at successive positions determined by the escapement of the characters imaged.
DDIF$K_FMT_CENTER_OF_PATH	The length of text strings, as given by the sum of the character escapements, is subtracted from the length of the path, and the remaining space is evenly distributed between the first character and the start of the path, and the last character and the end of the path.
DDIF$K_FMT_FLUSH_PATH_END	The text string is imaged such that the right alignment point of the last character is aligned with the end of the text string when normal escapement is applied.
DDIF$K_FMT_FLUSH_PATH_BOTH	The text string is imaged such that the left alignment point of the first character is aligned with the start of the text path, and the right alignment point of the last character is aligned with the end of the path.

6.20.21.5.4 Margin Frame Parameters

The margin frame parameters (selected by specifying DDIF$_SGA_FRM_POSITION_C as DDIF$K_FRAME_MARGIN) are specified using the following items:

- A margin base offset indicator (type DDIF$_SGA_FRMMAR_BASE_OFFSET_C) that indicates whether the base offset is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A margin base offset item (type DDIF$_SGA_FRMMAR_BASE_OFFSET) that specifies the vertical offset from the current baseline for the lower edge of the frame. This item is encoded as a variable.
- A margin near offset indicator (type DDIF$_SGA_FRMMAR_NEAR_OFFSET_C) that indicates whether the horizontal offset is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A margin near offset item (type DDIF$_SGA_FRMMAR_NEAR_OFFSET) that specifies the horizontal offset from the side of the frame nearest the reference frame to the corresponding side of the reference frame. This item is encoded as a variable.

- A margin horizontal item (type DDIF$_SGA_FRMMAR_HORIZONTAL) that defines a standard or private label that specifies the preferred horizontal position of the lower left corner of the frame. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_FRMMAR_CLOSEST_EDGE	The position of the frame depends on the page side. If the page is a left page, the frame is positioned to the left of the left-most galley; if the page is a right page, the frame is positioned to the right of the right-most galley.
DDIF$K_FRMMAR_FURTHEST_EDGE	The frame is positioned opposite the page side. If the page is a left page, the frame is positioned to the right of the right-most galley; if the page is a right page, the frame is positioned to the left of the left-most galley.
DDIF$K_FRMMAR_LEFT	The frame is positioned so that it is to the left of the left-most galley.
DDIF$K_FRMMAR_RIGHT	The frame is positioned so that it is to the right of the right-most galley.

6.20.21.6 Frame Content Transformation

The optional frame content transformation item (type DDIF$_SGA_FRM_TRANSFORM) specifies a transformation to be applied to the coordinates of content element within the frame, but not to the clipping region, outline, or other parameters associated with the frame. This item is encoded as a sequence of DDIF$_TRN aggregates. For more information on the DDIF$_TRN aggregate, see Section 6.32.

Frame content transformations are normally used when it is desirable to keep the coordinates of the content untransformed while providing the ability to view the content under different transformations. This avoids using repeated transformations on the content that would have the effect of altering the precision of the coordinates due to arithmetic roundoff during matrix multiplication.

6.20.22 Item Change List

The segment attributes aggregate supplies an item-change-list item (type DDIF$_SGA_ITEM_CHANGE_LIST) that specifies which attributes, as defined in this segment attributes aggregate, are explicitly defined at this segment level. That is, the item change list is an array of item codes that correspond to those items that are specifically defined in the segment attributes aggregate. Items that are inherited at this level from either default DDIF values (supplied by the CDA Toolkit), or from attributes defined at higher segment levels, are not referenced in this change list. Also, item codes of empty attributes are not included as part of this list.

Specifically, those item codes that return a status of CDA$_NORMAL in response to a call to the LOCATE ITEM routine make up this item change list. By using the item change list, an application can locate only those items in the segment attributes aggregate that are explicitly specified and interesting to the application.

The item change list item (type DDIF$_SGA_ITEM_CHANGE_LIST) is encoded as an array of type longword, where each longword contains the item code of the corresponding attribute items that are specified on this segment. This item is only valid if DDIF$_INHERIT_ATTRIBUTES is specified as a processing option.

6.20.23 Segment Attribute Items and Types

The previous sections discussed the various groups of related segment attributes. Table 6–29 lists all of the items in the segment attributes aggregate and their corresponding encodings.

Table 6–29 Segment Attributes Aggregate (DDIF$_SGA)

Item Name	Item Encoding
DDIF$_SGA_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates
DDIF$_SGA_CONTENT_STREAMS	Array of type string
DDIF$_SGA_CONTENT_CATEGORY	String with *add-info*
DDIF$_SGA_SEGMENT_TAGS	Array of type string with *add-info*
DDIF$_SGA_BINDING_DEFNS	Sequence of DDIF$_SGB aggregates
DDIF$_SGA_COMPUTE_C	Enumeration
DDIF$_SGA_CPTCPY_TARGET	String
DDIF$_SGA_CPTCPY_ERF_INDEX	Integer
DDIF$_SGA_CPTVAR_VARIABLE	String
DDIF$_SGA_CPTXRF_TARGET	String
DDIF$_SGA_CPTXRF_ERF_INDEX	Integer
DDIF$_SGA_CPTXRF_VARIABLE	String
DDIF$_SGA_CPTFNC_NAME	String
DDIF$_SGA_CPTFNC_PARAMETERS	Sequence of DDIF$_PVT aggregates
DDIF$_SGA_STRUCTURE_DESC_C	Enumeration
DDIF$_SGA_STRUCTURE_DESC	Sequence of DDIF$_OCC aggregates
DDIF$_SGA_LANGUAGE	Integer
DDIF$_SGA_LEGEND_UNIT_N	Integer
DDIF$_SGA_LEGEND_UNIT_D	Integer
DDIF$_SGA_LEGEND_UNIT_NAME	Array of type character string
DDIF$_SGA_UNITS_PER_MEASURE	Integer
DDIF$_SGA_UNITS_NAME	Array of type character string
DDIF$_SGA_ALT_PRESENTATION	Array of type character string
DDIF$_SGA_LAYOUT_C	Enumeration
DDIF$_SGA_LAYGLY_WRAP	Handle of DDIF$_LW1 aggregate
DDIF$_SGA_LAYGLY_LAYOUT	Handle of DDIF$_LL1 aggregate

(continued on next page)

Table 6–29 (Cont.) Segment Attributes Aggregate (DDIF$_SGA)

Item Name	Item Encoding
DDIF$_SGA_LAYPTH_PATH	Sequence of DDIF$_PTH aggregates
DDIF$_SGA_LAYPTH_FORMAT	Enumeration
DDIF$_SGA_LAYPTH_ORIENTATION_C	Enumeration
DDIF$_SGA_LAYPTH_ORIENTATION	Variable
DDIF$_SGA_LAYPTH_H_ALIGN	Enumeration
DDIF$_SGA_LAYPTH_V_ALIGN	Enumeration
DDIF$_SGA_LAYREL_H_RATIO_N	Integer
DDIF$_SGA_LAYREL_H_RATIO_D	Integer
DDIF$_SGA_LAYREL_H_CONSTANT_C	Measurement enumeration
DDIF$_SGA_LAYREL_H_CONSTANT	Variable
DDIF$_SGA_LAYREL_V_RATIO_N	Integer
DDIF$_SGA_LAYREL_V_RATIO_D	Integer
DDIF$_SGA_LAYREL_V_CONSTANT_C	Measurement enumeration
DDIF$_SGA_LAYREL_V_CONSTANT	Variable
DDIF$_SGA_LAYPOS_TEXT_POSITION	Enumeration
DDIF$_SGA_FONT_DEFNS	Sequence of DDIF$_FTD aggregates
DDIF$_SGA_PATTERN_DEFNS	Sequence of DDIF$_PTD aggregates
DDIF$_SGA_PATH_DEFNS	Sequence of DDIF$_PHD aggregates
DDIF$_SGA_LINE_STYLE_DEFNS	Sequence of DDIF$_LSD aggregates
DDIF$_SGA_CONTENT_DEFNS	Sequence of DDIF$_CTD aggregates
DDIF$_SGA_TYPE_DEFNS	Sequence of DDIF$_TYD aggregates
DDIF$_SGA_TXT_MASK_PATTERN	Integer
DDIF$_SGA_TXT_FONT	Integer
DDIF$_SGA_TXT_RENDITION	Array of type enumeration
DDIF$_SGA_TXT_HEIGHT_C	Measurement enumeration
DDIF$_SGA_TXT_HEIGHT	Variable
DDIF$_SGA_TXT_SET_SIZE_N	Integer
DDIF$_SGA_TXT_SET_SIZE_D	Integer
DDIF$_SGA_TXT_DIRECTION	Enumeration
DDIF$_SGA_TXT_DEC_ALIGNMENT	Array of type character string
DDIF$_SGA_TXT_LEADER_SPACE_C	Measurement enumeration
DDIF$_SGA_TXT_LEADER_SPACE	Variable
DDIF$_SGA_TXT_LEADER_BULLET	Character string
DDIF$_SGA_TXT_LEADER_ALIGN	Enumeration
DDIF$_SGA_TXT_LEADER_STYLE	Enumeration
DDIF$_SGA_TXT_PAIR_KERNING	Boolean

(continued on next page)

Table 6–29 (Cont.) Segment Attributes Aggregate (DDIF$_SGA)

Item Name	Item Encoding
DDIF$_SGA_LIN_WIDTH_C	Measurement enumeration
DDIF$_SGA_LIN_WIDTH	Variable
DDIF$_SGA_LIN_STYLE	Integer
DDIF$_SGA_LIN_PATTERN_SIZE_C	Measurement enumeration
DDIF$_SGA_LIN_PATTERN_SIZE	Variable
DDIF$_SGA_LIN_MASK_PATTERN	Integer
DDIF$_SGA_LIN_END_START	Enumeration
DDIF$_SGA_LIN_END_FINISH	Enumeration
DDIF$_SGA_LIN_END_SIZE_C	Measurement enumeration
DDIF$_SGA_LIN_END_SIZE	Variable
DDIF$_SGA_LIN_JOIN	Enumeration
DDIF$_SGA_LIN_MITER_LIMIT_N	Integer
DDIF$_SGA_LIN_MITER_LIMIT_D	Integer
DDIF$_SGA_LIN_INTERIOR_PATTERN	Integer
DDIF$_SGA_MKR_STYLE	Enumeration
DDIF$_SGA_MKR_MASK_PATTERN	Integer
DDIF$_SGA_MKR_SIZE_C	Measurement enumeration
DDIF$_SGA_MKR_SIZE	Variable
DDIF$_SGA_GLY_ATTRIBUTES	Handle of DDIF$_GLA aggregate
DDIF$_SGA_IMG_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates
DDIF$_SGA_IMG_PIXEL_PATH	Integer
DDIF$_SGA_IMG_LINE_PROGRESSION	Integer
DDIF$_SGA_IMG_PP_PIXEL_DIST	Integer
DDIF$_SGA_IMG_LP_PIXEL_DIST	Integer
DDIF$_SGA_IMG_BRT_POLARITY	Enumeration
DDIF$_SGA_IMG_GRID_TYPE	Enumeration
DDIF$_SGA_IMG_TIMING_DESC	Binary relative time
DDIF$_SGA_IMG_SPECTRAL_MAPPING	Enumeration
DDIF$_SGA_IMG_LOOKUP_TABLES_C	Enumeration
DDIF$_SGA_IMG_LOOKUP_TABLES	Variable
DDIF$_SGA_IMG_COMP_ WAVELENGTH_C	Enumeration
DDIF$_SGA_IMG_COMP_WAVELENGTH	Variable
DDIF$_SGA_IMG_COMP_SPACE_ORG	Enumeration
DDIF$_SGA_IMG_PLANES_PER_PIXEL	Integer
DDIF$_SGA_IMG_PLANE_SIGNIF	Enumeration

(continued on next page)

Table 6–29 (Cont.) Segment Attributes Aggregate (DDIF$_SGA)

Item Name	Item Encoding
DDIF$_SGA_IMG_NUMBER_OF_COMP	Integer
DDIF$_SGA_IMG_BITS_PER_COMP	Array of type integer
DDIF$_SGA_FRM_FLAGS	Longword
DDIF$_SGA_FRM_BOX_LL_X_C	Measurement enumeration
DDIF$_SGA_FRM_BOX_LL_X	Variable
DDIF$_SGA_FRM_BOX_LL_Y_C	Measurement enumeration
DDIF$_SGA_FRM_BOX_LL_Y	Variable
DDIF$_SGA_FRM_BOX_UR_X_C	Measurement enumeration
DDIF$_SGA_FRM_BOX_UR_X	Variable
DDIF$_SGA_FRM_BOX_UR_Y_C	Measurement enumeration
DDIF$_SGA_FRM_BOX_UR_Y	Variable
DDIF$_SGA_FRM_OUTLINE	Sequence of DDIF$_PTH aggregates
DDIF$_SGA_FRM_CLIPPING	Sequence of DDIF$_PTH aggregates
DDIF$_SGA_FRM_POSITION_C	Enumeration
DDIF$_SGA_FRMFXD_POSITION_X_C	Measurement enumeration
DDIF$_SGA_FRMFXD_POSITION_X	Variable
DDIF$_SGA_FRMFXD_POSITION_Y_C	Measurement enumeration
DDIF$_SGA_FRMFXD_POSITION_Y	Variable
DDIF$_SGA_FRMINL_BASE_OFFSET_C	Measurement enumeration
DDIF$_SGA_FRMINL_BASE_OFFSET	Variable
DDIF$_SGA_FRMGLY_VERTICAL	Enumeration
DDIF$_SGA_FRMGLY_HORIZONTAL	Enumeration
DDIF$_SGA_FRMMAR_BASE_OFFSET_C	Measurement enumeration
DDIF$_SGA_FRMMAR_BASE_OFFSET	Variable
DDIF$_SGA_FRMMAR_NEAR_OFFSET_C	Measurement enumeration
DDIF$_SGA_FRMMAR_NEAR_OFFSET	Variable
DDIF$_SGA_FRMMAR_HORIZONTAL	Enumeration
DDIF$_SGA_FRM_TRANSFORM	Sequence of DDIF$_TRN aggregates
DDIF$_SGA_ITEM_CHANGE_LIST	Item change list

6.21 Content Definition

The content definition aggregate (type DDIF$_CTD) lets you specify a labeled generic content definition that can be referenced by nested segments. The content definition can be specified using any one of the following items:

- A content label item (type DDIF$_CTD_LABEL) that specifies the label by which the content is referenced. This item is encoded as a string.

- An optional content external target item (type DDIF$_CTD_EXTERNAL_TARGET) that specifies the label of the segment being referenced. This item is encoded as a string. If it is not specified, the entire document is being referenced.
- An optional external reference index item (type DDIF$_CTD_EXTERNAL_ERF_INDEX) that specifies an index into a list of external references stored in the document header. This item is encoded as an integer. If it is not specified, the reference is to the current document.
- An optional content value item (type DDIF$_CTD_VALUE) that specifies the content elements. This item is encoded as a sequence of content, which can be the handle of any of the following aggregates:

DDIF$_ARC	DDIF$_BEZ	DDIF$_CRF
DDIF$_EXT	DDIF$_FAS	DDIF$_GRP
DDIF$_GTX	DDIF$_HRD	DDIF$_IMG
DDIF$_LIN	DDIF$_PVT	DDIF$_SEG
DDIF$_SFT	DDIF$_TXT	

- An optional content private data item (type DDIF$_CTD_PRIVATE_DATA) that specifies the private data associated with the definition. This item is encoded as a sequence of DDIF$_PVT aggregates. For more information on the DDIF$_PVT aggregate, see Section 6.15.2.

Table 6–30 lists all the items in a content definition aggregate and their corresponding encodings.

Table 6–30 Content Definition Aggregate (DDIF$_CTD)

Item Name	Item Encoding
DDIF$_CTD_LABEL	String
DDIF$_CTD_EXTERNAL_TARGET	String
DDIF$_CTD_EXTERNAL_ERF_INDEX	Integer
DDIF$_CTD_VALUE	Sequence of content
DDIF$_CTD_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates

6.22 Font Definition

The font definition aggregate (type DDIF$_FTD) defines a font for use within a segment. This aggregate contains the following items:

- A font number item (type DDIF$_FTD_NUMBER) that is used to reference the font within the defining segment. This item is encoded as an integer.
- A font identifier item (type DDIF$_FTD_IDENTIFIER) that specifies a font name. This item is encoded as a string.

- An optional font private data item (type DDIF$_FTD_PRIVATE_DATA) that specifies the private data associated with the definition. This item is encoded as a sequence of DDIF$_PVT aggregates. For more information on the DDIF$_PVT aggregate, see Section 6.15.2.

Table 6–31 lists all the items in a font definition aggregate and their corresponding encodings.

Table 6–31 Font Definition Aggregate (DDIF$_FTD)

Item Name	Item Encoding
DDIF$_FTD_NUMBER	Integer
DDIF$_FTD_IDENTIFIER	String
DDIF$_FTD_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates

6.23 Line-Style Definition

The line-style definition aggregate (type DDIF$_LSD) models the description of a line-style pattern for reference within the assigned scope. The line-style definition aggregate contains the following items:

- A line-style number (type DDIF$_LSD_NUMBER) that specifies a number by which the defined line style is referenced from within the scope of the definition. This item is encoded as an integer.
- A line-style pattern item (type DDIF$_LSD_PATTERN) that specifies the line-style pattern being defined. This item is encoded as an array of type integer.

  Each bit in each integer value provided is interpreted to determine the state of the corresponding pixel. For example, the following longword value would correspond to the line pattern shown.

```
11110001111000111100011110001111
```

  This value for DDIF$_LSD_PATTERN produces the following line-style pattern:

```
"----    ----    ----    ----    ----"
```

If your pattern must be defined using more than 32 bits, you must use additional longwords in an array to specify the pattern. For each longword specified, you must increment the aggregate index by 1. The initial value of the aggregate index is 0.

- An optional line-style private data item (type DDIF$_LSD_PRIVATE_DATA) that specifies the private data associated with the definition. This item is encoded as a sequence of DDIF$_PVT aggregates. For more information on the DDIF$_PVT aggregate, see Section 6.15.2.

Table 6–32 lists all the items in the line-style definition aggregate and their corresponding encodings.

Table 6–32 Line-Style Definition Aggregate (DDIF$_LSD)

Item Name	Item Encoding
DDIF$_LSD_NUMBER	Integer
DDIF$_LSD_PATTERN	Array of type integer
DDIF$_LSD_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates

6.24 Path Definition

The path definition aggregate (type DDIF$_PHD) models the description of a composite path for reference within the assigned scope. The path definition aggregate contains the following items:

- A path number item (type DDIF$_PHD_NUMBER) that specifies a number by which the defined path is referenced from within the scope of the definition. This item is encoded as an integer.
- A path description item (type DDIF$_PHD_DESCRIPTION) that specifies the composite path being defined. This item is encoded as a sequence of DDIF$_PTH aggregates. For more information on the DDIF$_PTH aggregate, see Section 6.19.
- An optional path private data item (type DDIF$_PHD_PRIVATE_DATA) that specifies the private data associated with the definition. This item is encoded as a sequence of DDIF$_PVT aggregates. For more information on the DDIF$_PVT aggregate, see Section 6.15.2.

Table 6–33 lists all the items in the path definition aggregate and their corresponding encodings.

Table 6–33 Path Definition Aggregate (DDIF$_PHD)

Item Name	Item Encoding
DDIF$_PHD_NUMBER	Integer
DDIF$_PHD_DESCRIPTION	Sequence of DDIF$_PTH aggregates
DDIF$_PHD_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates

6.25 Pattern Definition

The pattern definition aggregate (type DDIF$_PTD) contains the following items:

- A pattern number item (type DDIF$_PTD_NUMBER) that specifies a number by which the pattern is referenced. This item is encoded as an integer. In addition to the user-defined pattern numbers, several predefined patterns are provided. These patterns are described in Appendix F.
- A pattern definition indicator (type DDIF$_PTD_DEFN_C) that selects the definition of the pattern as either a solid color or a standard pattern. This item is encoded as an enumeration. Valid values are as follows:

  DDIF$K_SOLID_COLOR — Indicates a predefined solid fill pattern, assigned a single color. If this value is specified, you must supply values for the items DDIF$_PTD_SOL_COLOR_C through DDIF$_PTD_SOL_COLOR_B.

  DDIF$K_STANDARD_PATTERN — Indicates a reference to a standard pattern and a color map for it. The color map is defined in terms of previously defined solid patterns. If this value is specified, you must supply values for the items DDIF$_PTD_PAT_NUMBER and DDIF$_PTD_PAT_COLORS.

  DDIF$K_RASTER_PATTERN — Indicates an image data unit that represents the pattern. If this value is specified, you must supply a value for the item DDIF$_PTD_RAS_PATTERN.

- An optional solid color indicator (type DDIF$_PTD_SOL_COLOR_C) that must be completed if DDIF$_PTD_DEFN_C was specified as DDIF$K_SOLID_COLOR. This item is encoded as an enumeration. Valid values are as follows:

  DDIF$K_RGB_COLOR — Indicates that red/green/blue colors are available. If you specify this color type, you must supply values for the items DDIF$_PTD_SOL_COLOR_R through DDIF$_PTD_SOL_COLOR_B.

  DDIF$K_TRANSPARENCY — Indicates that colors are not available. If you specify this color type, you should not supply any values for the additional background color items.

- A red intensity item (type DDIF$_PTD_SOL_COLOR_R) that indicates the level of red intensity. This item is encoded as a single-precision floating-point value in the range of 0.0 to 1.0.
- A green intensity item (type DDIF$_PTD_SOL_COLOR_G) that indicates the level of green intensity. This item is encoded as a single-precision floating-point value in the range of 0.0 to 1.0.
- A blue intensity item (type DDIF$_PTD_SOL_COLOR_B) that indicates the level of blue intensity. This item is encoded as a single-precision floating-point value in the range of 0.0 to 1.0.
- A standard pattern number item (type DDIF$_PTD_PAT_NUMBER) that must be completed if DDIF$_PTD_DEFN_C was specified as DDIF$K_STANDARD_PATTERN. This item specifies the number of a standard pattern selected from the available patterns. This item is encoded as an integer. The standard patterns consist of pixel masks. Pixels are imaged in the indicated pattern color, according to the pixel values.

  You must select one of the predefined pattern numbers for this item. These patterns are described in Appendix F.
- A pattern colors item (type DDIF$_PTD_PAT_COLORS) that must be completed if DDIF$_PTD_DEFN_C was specified as DDIF$K_STANDARD_PATTERN. This item specifies a sequence of colors that form the color map for the pattern mask. This item is encoded as an array of type integer.

  The sequence of colors models an array in which the color of each entry maps to the number formed by the corresponding bit pattern in the pattern definition. A single bit-plane pattern mask has two colors, while a two-plane pattern has four. The significance of bits in the bit plane is specified along with the standard pattern definitions.
- A raster pattern item (type DDIF$_PTD_RAS_PATTERN) that must be completed if DDIF$_PTD_DEFN_C was specified as DDIF$K_RASTER_PATTERN. This item specifies the image data unit that represents the pattern. This item is encoded as the handle of a DDIF$_IDU aggregate. For more information on the DDIF$_IDU aggregate, see Section 6.18.
- An optional pattern private data item (type DDIF$_PTD_PRIVATE_DATA) that specifies the private data associated with the definition. This item is encoded as a sequence of DDIF$_PVT aggregates. For more information on the DDIF$_PVT aggregate, see Section 6.15.2.

Table 6–34 lists all the items in the pattern definition aggregate and their corresponding encodings.

Table 6–34 Pattern Definition Aggregate (DDIF$_PTD)

Item Name	Item Encoding
DDIF$_PTD_NUMBER	Integer
DDIF$_PTD_DEFN_C	Enumeration
DDIF$_PTD_SOL_COLOR_C	Enumeration
DDIF$_PTD_SOL_COLOR_R	Single-precision floating-point
DDIF$_PTD_SOL_COLOR_G	Single-precision floating-point
DDIF$_PTD_SOL_COLOR_B	Single-precision floating-point
DDIF$_PTD_PAT_NUMBER	Integer
DDIF$_PTD_PAT_COLORS	Array of type integer
DDIF$_PTD_RAS_PATTERN	Handle of DDIF$_IDU aggregate
DDIF$_PTD_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates

6.26 Segment Binding

The segment binding aggregate (type DDIF$_SGB) defines a variable by its name, and defines the method used to calculate its value. This aggregate contains the following items:

- A variable name item (type DDIF$_SGB_VARIABLE_NAME) that specifies the name of the variable being defined. This item is encoded as a string.
- A variable value indicator (type DDIF$_SGB_VARIABLE_VALUE_C) that indicates the type of variable value. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_COUNTER_VARIABLE	A variable that counts occurrences of nested segments with a specified tag, or occurrences of designated types of layout objects within nested segments. Note that the value of a counter variable varies within the segment, and cannot be cross-referenced from outside the segment. However, its value can at some point be captured in the definition of computed variables, which can be cross-referenced if the segment has a segment identifier. If you specify this value, you must supply values for the items DDIF$_SGB_CTR_TRIGGER_C through DDIF$_SGB_CTR_TYPE.
DDIF$K_COMPUTED_VARIABLE	A variable that has a constant value throughout the segment; its value is the value of the expression at the point of definition. If you specify this value, you must supply values for the items DDIF$_SGB_COM_STRING_EXPR_C and DDIF$_SGB_COM_STRING_EXPR.

DDIF$K_LIST_VARIABLE	A variable that contains an array of records. If you specify this value, you must supply a value for the item DDIF$_SGB_RCD_LIST.

Each of these types of variable values is discussed in the following sections, along with its corresponding aggregate items.

6.26.1 Counter Variable Values

Counter variable values (selected by specifying DDIF$_SGB_VARIABLE_VALUE_C as DDIF$K_COUNTER_VARIABLE) are described using the following items:

- An optional counter trigger indicator (type DDIF$_SGB_CTR_TRIGGER_C) that indicates the type of object to be counted. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_TAGGED_SEGMENT_TRIGGER	Counts tagged segments. In this case, the DDIF$_SGB_CTR_TRIGGER item is encoded as a string.
DDIF$K_LAYOUT_OBJECT_TRIGGER	Counts layout objects. In this case, the DDIF$_SGB_CTR_TRIGGER item is encoded as an enumeration that can accept any one of the following values:

DDIF$K_DOCUMENT_LAYOUT_OBJECT	Specifies that document layout objects are to be counted.
DDIF$K_PAGE_SET_LAYOUT_OBJECT	Specifies that page set layout objects are to be counted.
DDIF$K_PAGE_LAYOUT_OBJECT	Specifies that page layout objects are to be counted.
DDIF$K_FRAME_LAYOUT_OBJECT	Specifies that frame layout objects are to be counted.
DDIF$K_BLOCK_LAYOUT_OBJECT	Specifies that block layout objects are to be counted.
DDIF$K_LINE_LAYOUT_OBJECT	Specifies that line layout objects are to be counted.

- A counter trigger item (type DDIF$_SGB_CTR_TRIGGER) that specifies the object to be counted. This item is encoded as a variable.
- A counter initialization indicator (type DDIF$_SGB_CTR_INIT_C) that indicates the method used to express the initial value for the counter. This item is encoded as an expression enumeration.
- A counter initialization item (type DDIF$_SGB_CTR_INIT) that specifies the initial value for the counter. This item is encoded as a variable. The default value for this item is 1.

- An optional counter style item (type DDIF$_SGB_CTR_STYLE) that determines how the counter value should be converted to text for display. This item is encoded as a sequence of DDIF$_CTS aggregates. For more information on the DDIF$_CTS aggregate, see Section 6.28.
- A counter type item (type DDIF$_SGB_CTR_TYPE) that determines how nested occurrences of counted objects should be displayed, and on what conditions the counter should be reset to its initial value. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_MILITARY_COUNTER	All variables of this name in the current and parent segments are displayed, separated by text.
DDIF$K_OFFICE_COUNTER	Only the value of the variable in the current segment is displayed.
DDIF$K_PAGE_RELATIVE_ COUNTER	This style is never hierarchical, and is reset for every page. Footnote numbering on a per-page basis is an example of page-relative counting.

6.26.2 Computed Variable Values

Computed variable values (selected by specifying DDIF$_SGB_ VARIABLE_VALUE_C as DDIF$K_COMPUTED_VARIABLE) are described using the following items:

- A computed string expression indicator (type DDIF$_SGB_COM_ STRING_EXPR_C) that indicates whether an element of the expression is a text constant or a string representation. This item is encoded as an array of type enumeration. Valid values are as follows:

DDIF$K_TEXT_ELEMENT	An element of the expression is a text constant. In this case, DDIF$_SGB_COM_ STRING_EXPR is encoded as a character string.
DDIF$K_VARIABLE_ELEMENT	An element of the expression is a string representation. In this case, DDIF$_SGB_ COM_STRING_EXPR is encoded as a string.

- A computed string expression item (type DDIF$_SGB_COM_STRING_ EXPR) that specifies the string expression. This item is encoded as an array of type variable.

6.26.3 List Variable Values

List variable values (selected by specifying DDIF$_SGB_VARIABLE_ VALUE_C as DDIF$K_LIST_VARIABLE) are described using a record list item (type DDIF$_SGB_RCD_LIST). This item defines a record structure that consists of one or more primitive data types, expressed as references to variables. This item is encoded as a sequence of DDIF$_ RCD aggregates. For more information on the DDIF$_RCD aggregate, see Section 6.30.

6.26.4 Segment Binding Items and Types

The items described in the previous sections make up the segment binding aggregate. Table 6–35 lists all the items in the segment binding aggregate and their corresponding encodings.

Table 6–35 Segment Binding Aggregate (DDIF$_SGB)

Item Name	Item Encoding
DDIF$_SGB_VARIABLE_NAME	String
DDIF$_SGB_VARIABLE_VALUE_C	Enumeration
DDIF$_SGB_CTR_TRIGGER_C	Enumeration
DDIF$_SGB_CTR_TRIGGER	Variable
DDIF$_SGB_CTR_INIT_C	Expression enumeration
DDIF$_SGB_CTR_INIT	Variable
DDIF$_SGB_CTR_STYLE	Sequence of DDIF$_CTS aggregates
DDIF$_SGB_CTR_TYPE	Enumeration
DDIF$_SGB_COM_STRING_EXPR_C	Array of type enumeration
DDIF$_SGB_COM_STRING_EXPR	Array of type variable
DDIF$_SGB_RCD_LIST	Sequence of DDIF$_RCD aggregates

6.27 Type Definition

The segment type definition aggregate (type DDIF$_TYD) defines a labeled set of generic segment attributes for reference from nested segments. This aggregate contains the following items:

- A type label item (type DDIF$_TYD_LABEL) that specifies the label by which the type is referenced. This item is encoded as a string. If segment types with the same name are defined in the document, the most recent definition is used.
- An optional type parent item (type DDIF$_TYD_PARENT) that specifies the label of a segment type whose attributes are applied prior to applying the attributes of this type. This item is encoded as a string.
- An optional type attributes item (type DDIF$_TYD_ATTRIBUTES) that specifies the segment attributes that are applied to segments that reference the type being defined. This item is encoded as the handle of a DDIF$_SGA aggregate. For more information on the DDIF$_SGA aggregate, see Section 6.20.
- An optional type private data item (type DDIF$_TYD_PRIVATE_DATA) that specifies the private data associated with the definition. This item is encoded as a sequence of DDIF$_PVT aggregates. For more information on the DDIF$_PVT aggregate, see Section 6.15.2.

Table 6–36 lists all the items in the type definition aggregate and their corresponding encodings.

Table 6–36 Type Definition Aggregate (DDIF$_TYD)

Item Name	Item Encoding
DDIF$_TYD_LABEL	String
DDIF$_TYD_PARENT	String
DDIF$_TYD_ATTRIBUTES	Handle of DDIF$_SGA aggregate
DDIF$_TYD_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates

6.28 Counter Style

The counter style aggregate (type DDIF$_CTS) describes a display style to be used for counters. This aggregate contains the following items:

- A counter style indicator (type DDIF$_CTS_STYLE_C) that indicates the counter style to be used. This item is encoded as an enumeration. Valid values are as follows:

  DDIF$K_NUMBER_STYLE — The type of conversion used to present the variable as an alphanumeric string. In this case, the DDIF$_CTS_STYLE item is encoded as an enumeration that accepts any one of the following values:

DDIF$K_ARABIC_COUNTER	Arabic numbers
DDIF$K_L_ROMAN_COUNTER	Lowercase roman numerals
DDIF$K_U_ROMAN_COUNTER	Uppercase roman numerals
DDIF$K_L_LATIN_COUNTER	Lowercase Latin letters
DDIF$K_U_LATIN_COUNTER	Uppercase Latin letters
DDIF$K_W_ARABIC_COUNTER	Wide arabic numbers
DDIF$K_WL_ROMAN_COUNTER	Wide lowercase roman numerals
DDIF$K_WU_ROMAN_COUNTER	Wide uppercase roman numerals
DDIF$K_WL_LATIN_COUNTER	Wide lowercase Latin letters
DDIF$K_WU_LATIN_COUNTER	Wide uppercase Latin letters
DDIF$K_WK_50_COUNTER	Wide Katakana 50
DDIF$K_WK_IROHA_COUNTER	Wide Katakana Iroha
DDIF$K_HEBREW_COUNTER	Hebrew

  DDIF$K_BULLET_STYLE — An array of type character string, for which the counter value constitutes an index that selects the bullet. If the counter value exceeds the number of elements in the array, then the array is reused. In this case, the DDIF$_CTS_STYLE item is encoded as an array of type character string.

DDIF$K_STYLE_SEPARATOR	A constant text string added to the converted string as a value separator for the military style. In this case, the DDIF$_CTS_STYLE item is encoded as a character string.

- A counter style item (type DDIF$_CTS_STYLE) that contains the counter. This item is encoded as a variable.

Table 6–37 lists the items in the counter style aggregate and their corresponding encodings.

Table 6–37 Counter Style Aggregate (DDIF$_CTS)

Item Name	Item Encoding
DDIF$_CTS_STYLE_C	Enumeration
DDIF$_CTS_STYLE	Variable

6.29 Occurrence Definition

The occurrence definition aggregate (type DDIF$_OCC) describes the number of times the element of a structure definition can occur, and whether or not it can be omitted. This aggregate contains the following items:

- An occurrence indicator (type DDIF$_OCC_OCCURRENCE_C) that specifies the type of occurrence to be permitted. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_REQUIRED_OCCURRENCE	The construction must occur once and only once.
DDIF$K_OPTIONAL_OCCURRENCE	The construction can occur once or not at all.
DDIF$K_REPEAT_OCCURRENCE	The construction can occur one or more times.
DDIF$K_OPT_RPT_OCCURRENCE	The construction can occur zero or more times.

  There is no default or initial value for this aggregate item.

- A structure element indicator (type DDIF$_OCC_STRUCTURE_ELEMENT_C) that indicates whether a given element in the structure definition is the label of the referenced type or is a structure definition that is itself a defined substructure. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_SEQUENCE_STRUCTURE	Indicates a sequence of element occurrences that are constrained to occur in the order specified. In this case, the DDIF$_OCC_STRUCTURE_ELEMENT item is encoded as a sequence of DDIF$_OCC aggregates.

DDIF$K_SET_STRUCTURE	Indicates a set of element occurrences that are *not* constrained with respect to order. In this case, the DDIF$_OCC_STRUCTURE_ELEMENT item is encoded as a sequence of DDIF$_OCC aggregates.
DDIF$K_CHOICE_STRUCTURE	Indicates a group of element occurrences from which only one can be selected. In this case, the DDIF$_OCC_STRUCTURE_ELEMENT item is encoded as a sequence of DDIF$_OCC aggregates.
DDIF$K_REFERENCED_TYPE	Indicates the label assigned to the type reference whose occurrence in the document structure is being constrained. In this case, the DDIF$_OCC_STRUCTURE_ELEMENT item is encoded as a string.

- A structure item (type DDIF$_OCC_STRUCTURE_ELEMENT) that specifies the structure itself. This item is encoded as a variable.

Table 6–38 lists the elements in an occurrence definition aggregate and their corresponding encodings.

Table 6–38 Occurrence Definition Aggregate (DDIF$_OCC)

Item Name	Item Encoding
DDIF$_OCC_OCCURRENCE_C	Enumeration
DDIF$_OCC_STRUCTURE_ELEMENT_C	Enumeration
DDIF$_OCC_STRUCTURE_ELEMENT	Variable

6.30 Record Definition

The record definition aggregate (type DDIF$_RCD) defines a record structure that consists of one or more primitive data types, expressed as references to variables. Records are used in the calculation of computed content items, such as tables of figures and indexes. The record definition aggregate contains the following items:

- A record type item (type DDIF$_RCD_TYPE) that specifies the record type that will be applied to the variable when it is displayed. This item is encoded as a string.
- A record tag item (type DDIF$_RCD_TAG) that specifies an identifier that indicates which segments within the scope of this record definition cause the creation of a data record of this type. This item is encoded as a string.
- A record contents item (type DDIF$_RCD_CONTENTS) that specifies the variables of the record. Each variable name and its value at the segment in question become part of the record. This item is encoded as an array of type string.

Table 6–39 lists the items in a record definition aggregate and their corresponding encodings.

Table 6–39 Record Definition Aggregate (DDIF$_RCD)

Item Name	Item Encoding
DDIF$_RCD_TYPE	String
DDIF$_RCD_TAG	String
DDIF$_RCD_CONTENTS	Array of type string

6.31 Image Lookup Table Entry

The image (RGB) lookup table entry aggregate (type DDIF$_RGB) provides a method for creating a sequence of lookup table entries, where each entry describes a lookup table index that corresponds to the pixel that it maps. The RGB lookup table entry aggregate contains the following items:

- A lookup table index item (type DDIF$_RGB_LUT_INDEX) that specifies the integer value of the lookup-table-mapped pixel. This item is encoded as an integer that can range in value between 0 and $2^{16} - 1$.
- A lookup table red value item (type DDIF$_RGB_RED_VALUE) that specifies the red intensity value for the lookup-table-mapped pixel. This item is encoded as a single-precision floating-point value between 0.0 and 1.0.
- A lookup table green value item (type DDIF$_RGB_GREEN_VALUE) that specifies the green intensity value for the lookup-table-mapped pixel. This item is encoded as a single-precision floating-point value between 0.0 and 1.0.
- A lookup table blue value item (type DDIF$_RGB_BLUE_VALUE) that specifies the blue intensity value for the lookup-table-mapped pixel. This item is encoded as a single-precision floating-point value between 0.0 and 1.0.

Table 6–40 lists the items in the lookup table entry aggregate and their corresponding encodings.

Table 6–40 RGB Lookup Table Entry Aggregate (DDIF$_RGB)

Item Name	Item Encoding
DDIF$_RGB_LUT_INDEX	Integer
DDIF$_RGB_RED_VALUE	Single-precision floating-point
DDIF$_RGB_GREEN_VALUE	Single-precision floating-point
DDIF$_RGB_BLUE_VALUE	Single-precision floating-point

6.32 Transformation

The transformation aggregate (type DDIF$_TRN) provides mapping from one coordinate system to another. It provides the following capabilities:

- Asymmetric scaling
- Symmetric rotation or skewing of the axes
- Translation

The transformation aggregate contains the following items:

- A transformation parameter indicator (type DDIF$_TRN_PARAMETER_C) that indicates which parameter is being specified by DDIF$_TRN_PARAMETER. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_X_SCALE	Indicates the scale factor for *x*-coordinates. In this case, the DDIF$_TRN_PARAMETER item is encoded as a single-precision floating-point value.
DDIF$K_Y_SCALE	Indicates the scale factor for *y*-coordinates. In this case, the DDIF$_TRN_PARAMETER item is encoded as a single-precision floating-point value.
DDIF$K_X_TRANSLATE	Indicates translation values for *x*-coordinates. In this case, the DDIF$_TRN_PARAMETER item is encoded as a single-precision floating-point value.
DDIF$K_Y_TRANSLATE	Indicates translation values for *y*-coordinates. In this case, the DDIF$_TRN_PARAMETER item is encoded as a single-precision floating-point value.
DDIF$K_ROTATE	Indicates rotation of *x*- and *y*-coordinates. In this case, the DDIF$_TRN_PARAMETER item is encoded as a single-precision floating-point value.
DDIF$K_SKEW	Indicates a difference in rotation of *x*- and *y*-coordinates. In this case, the DDIF$_TRN_PARAMETER item is encoded as a single-precision floating-point value.
DDIF$K_MATRIX_2_BY_3	Indicates two columns of a 3x3 transformation matrix, specified in column order. Given 6 numbers in the order A-B-C-D-E-F, the matrix is as follows: A D 0 B E 0 C F 1 In this case, the DDIF$_TRN_PARAMETER item is encoded as an array (with 6 elements) of single-precision floating-point values.

DDIF$K_MATRIX_3_BY_3	Indicates a 3x3 transformation matrix, specified in column order. Given 9 numbers in the order A-B-C-D-E-F-G-H-I, the matrix is as follows: A D G B E H C F I In this case, the DDIF$_TRN_PARAMETER item is encoded as an array (with 9 elements) of single-precision floating-point values.

- A transformation parameter item (type DDIF$_TRN_PARAMETER) that contains the actual value of the translation parameter. This item is encoded as a variable.

Table 6–41 lists the items in the transformation aggregate and their corresponding encodings.

Table 6–41 Transformation Aggregate (DDIF$_TRN)

Item Name	Item Encoding
DDIF$_TRN_PARAMETER_C	Enumeration
DDIF$_TRN_PARAMETER	Variable

6.33 Generic Layout

The generic layout aggregate (type DDIF$_LG1) specifies a set of page descriptions along with rules about when to use a particular page description. It also enables you to describe a set of content descriptions that can be referenced from generic and/or specific pages to form content that appears on one or more pages.

The generic layout aggregate contains the following items:

- A private data item (type DDIF$_LG1_PRIVATE_DATA) that specifies nonstandard information associated with the generic layout descriptions. This item is encoded as a sequence of DDIF$_PVT aggregates. (For more information on the DDIF$_PVT aggregate, see Section 6.15.2.) The private data is typically used to associate names or relationships with the page and/or content descriptions.
- A page descriptions item (type DDIF$_LG1_PAGE_DESCRIPTIONS) that provides descriptions of actual page templates and rules for their use. This item is encoded as a sequence of DDIF$_PGD aggregates. For more information on the DDIF$_PGD aggregate, see Section 6.38.

Table 6–42 lists the items in the generic layout aggregate and their corresponding encodings.

Table 6–42 Generic Layout 1 Aggregate (DDIF$_LG1)

Item Name	Item Encoding
DDIF$_LG1_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates
DDIF$_LG1_PAGE_DESCRIPTIONS	Sequence of DDIF$_PGD aggregates

6.34 Specific Layout

The specific layout aggregate (type DDIF$_LS1) contains one page description for each page of the document, although pages that have identical layout can share a description for the sake of representational efficiency.

A document that contains specific layout can also have a generic layout specification, which is used to add new pages to the document. Specific page layouts can be derived from a generic layout, they can be manually generated, or they can be user-modified versions of layouts derived from generic layouts.

The specific layout of a document is represented as a list of page descriptions, or references to page descriptions that have been previously declared. The first specific page description is by default the first page, but you can override this by making selections within the content stream.

The specific layout aggregate contains the following items:

- A layout indicator (type DDIF$_LS1_LAYOUT_C) that indicates whether the layout is for a specific page or is a reference to a previously defined page. This item is encoded as an array of type enumeration. Valid values are as follows:

DDIF$K_SPECIFIC_PAGE	Indicates that the layout specified is the description of a specific page. In this case, the DDIF$_LS1_LAYOUT item is encoded as the handle of a DDIF$_PGD aggregate.
DDIF$K_REFERENCED_PAGE	Indicates that the layout specified is actually the label of a page layout description previously defined. In this case, the DDIF$_LS1_LAYOUT item is encoded as a string.

- A layout item (type DDIF$_LS1_LAYOUT) that defines the specific layout. This item is encoded as an array of type variable.

Note that the array items in each array must correspond. For example, if the first value in the layout indicator array specifies a referenced page, the first value in the layout array must contain a string specifying the label of the page layout description being referenced, and so on.

Table 6–43 lists the items in the specific layout aggregate and their corresponding encodings.

Table 6–43 Specific Layout 1 Aggregate (DDIF$_LS1)

Item Name	Item Encoding
DDIF$_LS1_LAYOUT_C	Array of type enumeration
DDIF$_LS1_LAYOUT	Array of type variable

6.35 Wrap Attributes

The wrap attributes aggregate (tpe DDIF$_LW1) contains the following items:

- An optional wrap format item (type DDIF$_LW1_WRAP_FORMAT) that specifies the format of text lines wrapped by the formatter. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_FMT_FLUSH_PATH_BEGIN	The first character is imaged at the start of the text path, and successive characters are imaged at successive positions determined by the escapement of the characters imaged.
DDIF$K_FMT_CENTER_OF_PATH	The length of text strings, as given by the sum of the character escapements, is subtracted from the length of the path, and the remaining space is evenly distributed between the first character and the start of the path, and the last character and the end of the path.
DDIF$K_FMT_FLUSH_PATH_END	The text string is imaged such that the right alignment point of the last character is aligned with the end of the text string when normal escapement is applied.
DDIF$K_FMT_FLUSH_PATH_BOTH	The text string is imaged such that the left alignment point of the first character is aligned with the start of the text path, and the right alignment point of the last character is aligned with the end of the path.

- An optional quad format item (type DDIF$_LW1_QUAD_FORMAT) that specifies the format of text lines that end in a hard (user-entered) new line. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_FMT_FLUSH_PATH_BEGIN	The first character is imaged at the start of the text path, and successive characters are imaged at successive positions determined by the escapement of the characters imaged.

DDIF$K_FMT_CENTER_OF_PATH	The length of text strings, as given by the sum of the character escapements, is subtracted from the length of the path, and the remaining space is evenly distributed between the first character and the start of the path, and the last character and the end of the path.
DDIF$K_FMT_FLUSH_PATH_END	The text string is imaged such that the right alignment point of the last character is aligned with the end of the text string when normal escapement is applied.
DDIF$K_FMT_FLUSH_PATH_BOTH	The text string is imaged such that the left alignment point of the first character is aligned with the start of the text path, and the right alignment point of the last character is aligned with the end of the path.

- An optional hyphenation flags item (type DDIF$_LW1_HYPHENATION_FLAGS) that specifies the Boolean parameters that affect hyphenation. This item is encoded as a longword. The possible values are as follows:

DDIF$M_HYPH_ALLOWED	If set, hyphenation is allowed in this segment.
DDIF$M_HYPH_PARAGRAPH	If set, the last line in the paragraph can end in a hyphen.
DDIF$M_HYPH_GALLEY_END	If set, hyphenation is allowed at the end of a galley.
DDIF$M_HYPH_PAGE_END	If set, words can be hyphenated across pages.
DDIF$M_HYPH_CAPITALIZED_WORD	If set, capitalized words can be hyphenated.

- An optional maximum hyphenation lines item (type DDIF$_LW1_MAXIMUM_HYPH_LINES) that specifies the maximum number of consecutive lines that can end with a hyphen. This item is encoded as an integer.
- An optional maximum orphan size (type DDIF$_LW1_MAXIMUM_ORPHAN_SIZE) that specifies the maximum orphan size. This item is encoded as an integer. This value specifies the maximum number of lines of text within the segment that can be left at the bottom of the galley if the rest of the lines are on the succeeding galley.
- An optional maximum widow size (type DDIF$_LW1_MAXIMUM_WIDOW_SIZE) that specifies the maximum widow size. This item is encoded as an integer. This value specifies the maximum number of lines of text within the segment that can be placed in the succeeding galley when the first line or lines are in the current galley.

Table 6–44 lists all the items in the wrap attributes aggregate and their corresponding encodings.

Table 6–44 Wrap Attributes 1 Aggregate (DDIF$_LW1)

Item Name	Item Encoding
DDIF$_LW1_WRAP_FORMAT	Enumeration
DDIF$_LW1_QUAD_FORMAT	Enumeration
DDIF$_LW1_HYPHENATION_FLAGS	Longword
DDIF$_LW1_MAXIMUM_HYPH_LINES	Integer
DDIF$_LW1_MAXIMUM_ORPHAN_SIZE	Integer
DDIF$_LW1_MAXIMUM_WIDOW_SIZE	Integer

6.36 Layout Attributes

The layout attributes aggregate (type DDIF$_LL1) contains the following items:

- An optional initial directive item (type DDIF$_LL1_INITIAL_DIRECTIVE) that forces a new line, galley, or page by means of a directive. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_DIR_NEW_PAGE	Begins a new page.
DDIF$K_DIR_NEW_LINE	Begins a new line of text.
DDIF$K_DIR_NEW_GALLEY	Begins a new layout galley (such as a column). Software that does not support galley layout interprets the new galley directive as a new page.
DDIF$K_DIR_TAB	Moves the horizontal text position to the next tab stop.
DDIF$K_DIR_SPACE	Specifies a space in the current font. The space directive is usually soft, and is used to indicate that software inserted a space between wrapped lines.
DDIF$K_DIR_HYPHEN_NEW_LINE	Specifies that the line break is preceded by a hyphen. This directive is typically soft, and is used to indicate that software inserted a hyphen at the place it broke the line.
DDIF$K_DIR_WORD_BREAK_POINT	Identifies an embedded point at which a word may be broken, if need be, for justification.
DDIF$K_DIR_LEADERS	Inserts leader characters according to the current leader attributes.
DDIF$K_DIR_BACKSPACE	Specifies that the first character following this directive should be centered over the last character imaged.

DDIF$K_NULL	Suppresses the inheritance of the initial-directive element of layout attributes. This directive has no effect on imaging or processing.
DDIF$K_DIR_NO_HYPHEN_WORD	Suppresses hyphenation until the next space character or space directive is encountered.

- An optional galley selection item (type DDIF$_LL1_GALLEY_SELECT) that forces the selection of a new layout galley by name. This item is encoded as a string.
- An optional pre-segment break condition item (type DDIF$_LL1_BREAK_BEFORE) that specifies the condition on which a break occurs before the segment. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_BREAK_ALWAYS	Always break to a new galley or page
DDIF$K_BREAK_NEVER	Never break to a new galley or page
DDIF$K_BREAK_IF_NEEDED	The formatter can break to a new galley or page at its discretion

- An optional in-segment break condition item (type DDIF$_LL1_BREAK_WITHIN) that specifies the condition on which a break occurs within a segment. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_BREAK_ALWAYS	Always break to a new galley or page
DDIF$K_BREAK_NEVER	Never break to a new galley or page
DDIF$K_BREAK_IF_NEEDED	The formatter can break to a new galley or page at its discretion

- An optional post-segment break condition item (type DDIF$_LL1_BREAK_AFTER) that specifies the condition on which a break occurs after the segment. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_BREAK_ALWAYS	Always break to a new galley or page
DDIF$K_BREAK_NEVER	Never break to a new galley or page
DDIF$K_BREAK_IF_NEEDED	The formatter can break to a new galley or page at its discretion

- An optional initial indent indicator (type DDIF$_LL1_INITIAL_INDENT_C) that specifies whether the initial indent value is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An optional initial indent item (type DDIF$_LL1_INITIAL_INDENT) that specifies the distance added to the current left indent to determine the minimum distance between the start of the path and the left alignment point of the first character in the text layout path. This item is encoded as a variable. The initial value is 0.
- An optional left indent indicator (type DDIF$_LL1_LEFT_INDENT_C) that indicates whether the left indent value is specified as a variable or constant value. This item is encoded as a measurement enumeration.

- An optional left indent item (type DDIF$_LL1_LEFT_INDENT) that specifies the distance added to the current left indent to create a new left indent, which determines the minimum distance between the start of the text layout path and the left alignment position of the first character on every wrapped line. This item is encoded as a variable. If no initial indent is specified, the left indent is used for the initial indent. The initial value of the left indent is 0. Note that the left indent inherited by a segment is the sum of the left indents specified by its parent segments.
- An optional right indent indicator (type DDIF$_LL1_RIGHT_INDENT_C) that indicates whether the right indent value is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An optional right indent item (type DDIF$_LL1_RIGHT_INDENT) that specifies the distance added to the current right indent to determine the new right indent, which is the minimum distance between the end of the text path and the last character imaged along the path. This item is encoded as a variable. The initial value of the right indent is 0. Note that the right indent inherited by a segment is the sum of the right indents specified by its parent segments.
- A space-before indicator (type DDIF$_LL1_SPACE_BEFORE_C) that indicates whether the space-before value is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A space-before item (type DDIF$_LL1_SPACE_BEFORE) that specifies the amount of space before the segment. This item is encoded as a variable with a default value of 0.
- A space-after indicator (type DDIF$_LL1_SPACE_AFTER_C) that indicates whether the space-after value is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A space-after item (type DDIF$_LL1_SPACE_AFTER) that specifies the amount of space after the segment. This item is encoded as a variable with a default value of 0.
- An optional leading ratio numerator item (type DDIF$_LL1_LEADING_RATIO_N) that specifies the magnitude of the escapement ratio to be used to increment or decrement the interline spacing in layout. This item is encoded as an integer. This ratio specifies the proportion of the normal line spacing used as "additional" line spacing. For example, a leading ratio of 1:1 doubles the total line spacing, and 2:1 triples it.
- An optional leading ratio denominator item (type DDIF$_LL1_LEADING_RATIO_D) that specifies the units of precision used in the escapement ratio that is used to increment or decrement the interline spacing in layout. This item is encoded as an integer.
- An optional leading constant indicator (type DDIF$_LL1_LEADING_CONSTANT_C) that indicates whether the interline spacing value is specified as a variable or constant value. This item is encoded as a measurement enumeration.

- An optional leading constant item (type DDIF$_LL1_LEADING_CONSTANT) that specifies the interline spacing value in the current measurement units. This item is encoded as a variable.
- An optional tab stops item (type DDIF$_LL1_TAB_STOPS) that specifies a sequence of fields along the current text path that cause text between tab directives to become aligned within the fields. This item is encoded as a sequence of DDIF$_TBS aggregates. For more information on the DDIF$_TBS aggregate, see Section 6.41.

Table 6–45 lists all the items in the layout attributes aggregate and their corresponding encodings.

Table 6–45 Layout Attributes 1 Aggregate (DDIF$_LL1)

Item Name	Item Encoding
DDIF$_LL1_INITIAL_DIRECTIVE	Enumeration
DDIF$_LL1_GALLEY_SELECT	String
DDIF$_LL1_BREAK_BEFORE	Enumeration
DDIF$_LL1_BREAK_WITHIN	Enumeration
DDIF$_LL1_BREAK_AFTER	Enumeration
DDIF$_LL1_INITIAL_INDENT_C	Measurement enumeration
DDIF$_LL1_INITIAL_INDENT	Variable
DDIF$_LL1_LEFT_INDENT_C	Measurement enumeration
DDIF$_LL1_LEFT_INDENT	Variable
DDIF$_LL1_RIGHT_INDENT_C	Measurement enumeration
DDIF$_LL1_RIGHT_INDENT	Variable
DDIF$_LL1_SPACE_BEFORE_C	Measurement enumeration
DDIF$_LL1_SPACE_BEFORE	Variable
DDIF$_LL1_SPACE_AFTER_C	Measurement enumeration
DDIF$_LL1_SPACE_AFTER	Variable
DDIF$_LL1_LEADING_RATIO_N	Integer
DDIF$_LL1_LEADING_RATIO_D	Integer
DDIF$_LL1_LEADING_CONSTANT_C	Measurement enumeration
DDIF$_LL1_LEADING_CONSTANT	Variable
DDIF$_LL1_TAB_STOPS	Sequence of DDIF$_TBS aggregates

6.37 Galley Attributes

The galley attributes aggregate (type DDIF$_GLA) lets you specify the characteristics of a galley that can be acquired from a generic galley definition or specified locally. This aggregate contains the following items:

- An optional galley top margin indicator (type DDIF$_GLA_TOP_MARGIN_C) that indicates whether the top margin is specified as a variable or constant value. This item is encoded as a measurement enumeration.

- An optional galley top margin item (type DDIF$_GLA_TOP_MARGIN) that specifies the distance from the top of the galley to the top of the topmost text line or frame displayed in the galley. This item is encoded as a variable. The initial value of this item is 0.
- An optional galley left margin indicator (type DDIF$_GLA_LEFT_MARGIN_C) that indicates whether the left margin is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An optional galley left margin item (type DDIF$_GLA_LEFT_MARGIN) that specifies the distance between the left side of the galley and the left side of the text lines and frames displayed in the galley. This item is encoded as a variable. The initial value of this item is 0.
- An optional galley right margin indicator (type DDIF$_GLA_RIGHT_MARGIN_C) that indicates whether the right margin is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An optional galley right margin item (type DDIF$_GLA_RIGHT_MARGIN) that specifies the distance between the right side of the galley and the right side of the text lines and frames displayed in the galley. This item is encoded as a variable. The initial value of this item is 0.
- An optional galley bottom margin indicator (type DDIF$_GLA_BOTTOM_MARGIN_C) that indicates whether the bottom margin is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- An optional galley bottom margin item (type DDIF$_GLA_BOTTOM_MARGIN) that specifies the distance from the bottom of the galley to the bottom of the lowest text line or frame displayed in the galley. This item is encoded as a variable. The initial value of this item is 0.

Table 6–46 lists all the items in the galley attributes aggregate and their corresponding encodings.

Table 6–46 Galley Attributes Aggregate (DDIF$_GLA)

Item Name	Item Encoding
DDIF$_GLA_TOP_MARGIN_C	Measurement enumeration
DDIF$_GLA_TOP_MARGIN	Variable
DDIF$_GLA_LEFT_MARGIN_C	Measurement enumeration
DDIF$_GLA_LEFT_MARGIN	Variable
DDIF$_GLA_RIGHT_MARGIN_C	Measurement enumeration

(continued on next page)

Table 6–46 (Cont.) Galley Attributes Aggregate (DDIF$_GLA)

Item Name	Item Encoding
DDIF$_GLA_RIGHT_MARGIN	Variable
DDIF$_GLA_BOTTOM_MARGIN_C	Measurement enumeration
DDIF$_GLA_BOTTOM_MARGIN	Variable

6.38 Page Description

The page description aggregate (type DDIF$_PGD) describes a page either as a single page layout or as a set of page layouts with conditions under which the different page layouts are used. A page layout is used when one of the galleys on the page is given text content. Galleys are connected to form a chain of successors used to format a flow of text. As each galley is invoked, the page on which it is described is invoked.

The page description aggregate contains the following items:

- A page description label item (type DDIF$_PGD_LABEL) that specifies the label by which the page description is referenced. This item is encoded as a string.
- An optional private data item (type DDIF$_PGD_PRIVATE_DATA) that allows for the inclusion of application-private data. This item is encoded as a sequence of DDIF$_PVT aggregates. For more information on the DDIF$_PVT aggregate, see Section 6.15.2.
- A page description indicator (type DDIF$_PGD_DESC_C) that indicates whether the page description is actually a set of page layouts or is a page layout defined for reference. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_PAGE_SET_DESC	A description of a set of page layouts, one of which is chosen based on the criteria presented in the page set. In this case, the DDIF$_PGD_DESC item is encoded as a sequence of DDIF$_PGS aggregates.
DDIF$K_PAGE_LAYOUT	A page layout description defined for reference from content or from page set descriptions. In this case, the DDIF$_PGD_DESC item is encoded as the handle of a DDIF$_PGL aggregate.

- A page description item (type DDIF$_PGD_DESC) that specifies the actual page description to be used. This item is encoded as a variable.

Table 6–47 lists all the items in the page description aggregate and their corresponding encodings.

Table 6–47 Page Description Aggregate (DDIF$_PGD)

Item Name	Item Encoding
DDIF$_PGD_LABEL	String
DDIF$_PGD_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates
DDIF$_PGD_DESC_C	Enumeration
DDIF$_PGD_DESC	Variable

6.39 Page Layout

The page layout aggregate (type DDIF$_PGL) describes a page, including its size, the galleys on the page, and any content specific to that particular page. The same page layout is shared by generic and specific layout.

The page layout aggregate contains the following items:

- A page layout identifier item (type DDIF$_PGL_LAYOUT_ID) that specifies a label used to reference the page layout. This item is encoded as a string.
- A page size nominal measure indicator (type DDIF$_PGL_SIZE_X_NOM_C) that indicates whether the nominal *x* measurement is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A page size nominal measure item (type DDIF$_PGL_SIZE_X_NOM) that specifies the nominal *x* measurement. This item is encoded as a variable.
- A page size *x* stretch indicator (type DDIF$_PGL_SIZE_X_STR_C) that indicates whether the *x* stretch amount is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A page size *x* stretch item (type DDIF$_PGL_SIZE_X_STR) that specifies the amount by which the *x* measurement can be extended. This item is encoded as a variable.
- A page size *x* shrink indicator (type DDIF$_PGL_SIZE_X_SHR_C) that indicates whether the *x* shrink amount is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A page size *x* shrink item (type DDIF$_PGL_SIZE_X_SHR) that specifies the amount by which the *x* measurement can be contracted. This item is encoded as a variable.
- A page size nominal measure indicator (type DDIF$_PGL_SIZE_Y_NOM_C) that indicates whether the nominal *y* measurement is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A page size nominal measure item (type DDIF$_PGL_SIZE_Y_NOM) that specifies the nominal *y* measurement. This item is encoded as a variable.

- A page size *y* stretch indicator (type DDIF$_PGL_SIZE_Y_STR_C) that indicates whether the *y* stretch amount is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A page size *y* stretch item (type DDIF$_PGL_SIZE_Y_STR) that specifies the amount by which the *y* measurement can be extended. This item is encoded as a variable.
- A page size *y* shrink indicator (type DDIF$_PGL_SIZE_Y_SHR_C) that indicates whether the *y* shrink amount is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A page size *y* shrink item (type DDIF$_PGL_SIZE_Y_SHR) that specifies the amount by which the *y* measurement can be contracted. This item is encoded as a variable.
- A page orientation item (type DDIF$_PGL_ORIENTATION) that defines the orientation of the page relative to the height and width. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_PORTRAIT_ORIENT	Portrait orientation puts the *y* axis along the height of the page.
DDIF$K_LANDSCAPE_ORIENT	Landscape orientation puts the *y* axis along the width of the page.

- An optional page prototype item (type DDIF$_PGL_PROTOTYPE) that specifies the label of the generic page description from which the layout being defined was derived. This item is encoded as a string. Any objects other than galleys in the page frame of the prototype definition are imaged in the new page layout.
- An optional page content item (type DDIF$_PGL_CONTENT) that must represent a frame whose origin is located at the lower lefthand corner of the frame. This item is encoded as a sequence of content. A sequence of content is a linked list of any of the following aggregate types:

DDIF$_ARC	DDIF$_BEZ	DDIF$_CRF
DDIF$_EXT	DDIF$_FAS	DDIF$_GRP
DDIF$_GTX	DDIF$_HRD	DDIF$_IMG
DDIF$_LIN	DDIF$_PVT	DDIF$_SEG
DDIF$_SFT	DDIF$_TXT	

  The DDIF$_PGL_CONTENT item contains the handle of the first aggregate in the sequence of content aggregates. Page content can reference definitions in the document content, but the document content cannot reference definitions in the page content. All page content coordinates are relative to the page coordinate system, but frames can be nested in the page content.

Table 6–48 lists all the items in the page layout aggregate and their corresponding encodings.

Table 6–48 Page Layout Aggregate (DDIF$_PGL)

Item Name	Item Encoding
DDIF$_PGL_LAYOUT_ID	String
DDIF$_PGL_SIZE_X_NOM_C	Measurement enumeration
DDIF$_PGL_SIZE_X_NOM	Variable
DDIF$_PGL_SIZE_X_STR_C	Measurement enumeration
DDIF$_PGL_SIZE_X_STR	Variable
DDIF$_PGL_SIZE_X_SHR_C	Measurement enumeration
DDIF$_PGL_SIZE_X_SHR	Variable
DDIF$_PGL_SIZE_Y_NOM_C	Measurement enumeration
DDIF$_PGL_SIZE_Y_NOM	Variable
DDIF$_PGL_SIZE_Y_STR_C	Measurement enumeration
DDIF$_PGL_SIZE_Y_STR	Variable
DDIF$_PGL_SIZE_Y_SHR_C	Measurement enumeration
DDIF$_PGL_SIZE_Y_SHR	Variable
DDIF$_PGL_ORIENTATION	Enumeration
DDIF$_PGL_PROTOTYPE	String
DDIF$_PGL_CONTENT	Sequence of content

6.40 Page Select

The page selection aggregate (type DDIF$_PGS) consists of one or more pages, one of which is selected based on the current formatting state. Each page selection aggregate consists of a pointer to a page in the list of page layouts, and the criteria that cause that particular page in the set to be selected.

The page select aggregate contains the following items:

- A page-side criteria item (type DDIF$_PGS_PAGE_SIDE_CRITERIA) that specifies the criteria for the side of the page that must be satisfied to use this page layout description. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_LEFT_PAGE	Used for left-hand pages when two pages are side by side. A page set that contains a left page must also contain a right page, and cannot contain a page specified as either page.
DDIF$K_RIGHT_PAGE	Used for right-hand pages when two pages are side by side. A page set that contains a right page must also contain a left page, and cannot contain a page specified as either page.
DDIF$K_EITHER_PAGE	The same page description is used for either left or right pages.

  The default is DDIF$K_EITHER_PAGE.

- A select page layout indicator (type DDIF$_PGS_SELECT_PAGE_LAYOUT_C) that indicates whether the selected page layout is specified by label or by definition. This item is encoded as an enumeration. Valid values are as follows:

DDIF$K_SELECT_BY_LABEL	Selects a page layout by specifying the label. In this case, the DDIF$_PGS_SELECT_PAGE_LAYOUT item is encoded as a string.
DDIF$K_SELECT_BY_DEFN	Selects a page layout by specifying its definition. In this case, the DDIF$_PGS_SELECT_PAGE_LAYOUT item is encoded as the handle of a DDIF$_PGL aggregate.

- A select page layout item (type DDIF$_PGS_SELECT_PAGE_LAYOUT) that specifies the selected page layout. This item is encoded as a variable.

Table 6–49 lists the items in the page select aggregate and their corresponding encodings.

Table 6–49 Page Select Aggregate (DDIF$_PGS)

Item Name	Item Encoding
DDIF$_PGS_PAGE_SIDE_CRITERIA	Enumeration
DDIF$_PGS_SELECT_PAGE_LAYOUT_C	Enumeration
DDIF$_PGS_SELECT_PAGE_LAYOUT	Variable

6.41 Tab Stop

The tab stop aggregate (type DDIF$_TBS) defines a set of fields along a text path. The tab stop measurements are always relative to the current path. A tab directive selects the next tab stop beyond the current text position in the current text direction. If no further tab stops are defined, the tab settings are repeated by adding the position of the last tab to each of the defined tab stops. All tab stops are relative to the beginning of the current path as defined by a galley or a string layout.

The tab stop aggregate contains the following items:

- A tab stop horizontal position indicator (type DDIF$_TBS_HORIZONTAL_POSITION_C) that indicates whether the horizontal position of the tab stop is specified as a variable or constant value. This item is encoded as a measurement enumeration.
- A tab stop horizontal position item (type DDIF$_TBS_HORIZONTAL_POSITION) that specifies the position of the tab stop relative to the origin of the current text path. This item is encoded as a variable.
- A tab stop type item (type DDIF$_TBS_TYPE) that specifies the type of tab stop alignment. This item is encoded as an enumeration and accepts any one of the following values:

DDIF$K_LEFT_TAB	The characters in the tab field are positioned with the left alignment point of the first character at the tab position.
DDIF$K_CENTER_TAB	The character following the tab directive is positioned such that the center alignment point is on the horizontal position of the tab stop.
DDIF$K_RIGHT_TAB	The string of characters is positioned such that the right alignment point of the last character is on the position of the right tab.
DDIF$K_DECIMAL_TAB	The first decimal point character subsequent to the tab directive is positioned such that the center alignment point of that character is at the horizontal position of the tab stop.

The default tab type is DDIF$K_LEFT_TAB.

- An optional tab stop leader item (type DDIF$_TBS_LEADER) that specifies an optional leader character to appear repeatedly between the tab directive in the document text and the character following the tab directive. This item is encoded as a character string.

  If no leader character is specified, none appears after that tab. The leader character is presented in the typeface and size attributes of the segment in which the tab directive occurs. Only one character can be specified.

Table 6–50 lists all the items in the tab stop aggregate and their corresponding encodings.

Table 6–50 Tab Stop Aggregate (DDIF$_TBS)

Item Name	Item Encoding
DDIF$_TBS_HORIZONTAL_POSITION_C	Measurement enumeration
DDIF$_TBS_HORIZONTAL_POSITION	Variable
DDIF$_TBS_TYPE	Enumeration
DDIF$_TBS_LEADER	Character string

CDA Reference Section

This section provides detailed discussions of the routines provided by the Compound Document Architecture toolkit.

CDA$AGGREGATE_TYPE_TO_OBJECT_ID AGGREGATE TYPE TO OBJECT ID

Translates a root aggregate type to an object identifier.

FORMAT

CDA$AGGREGATE_TYPE_TO_OBJECT_ID
aggregate-type ,buf-len ,buf-adr ,nam-len ,nam-buf, act-nam-len ,act-len

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

aggregate-type

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Type of the aggregate being translated. The **aggregate-type** argument is the address of an unsigned longword that specifies the root aggregate type. The root aggregate type must be DDIF$_DDF.

buf-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Length (in bytes) of the object identifier buffer. The **buf-len** argument is the address of an unsigned longword that specifies this buffer length.

buf-adr

VMS usage: **address**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the address of the object identifier. The **buf-adr** argument is the address of an unsigned longword that receives the address of an array of unsigned longwords that contains the object identifier.

nam-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Length (in bytes) of the domain name buffer. The **nam-len** argument is the address of an unsigned longword that contains the domain name buffer length.

nam-buf

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference, array reference**

Receives the address of the domain name buffer. The **nam-buf** argument is the address of an unsigned longword that receives the address of the domain name buffer.

act-nam-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the actual length (in bytes) of the domain name in the **nam-buf** buffer. The **act-nam-len** argument is the address of an unsigned longword that receives this length.

act-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the actual length (in bytes) of the object identifier. The **act-len** argument is the address of an unsigned longword that receives the object identifier length.

DESCRIPTION

The AGGREGATE TYPE TO OBJECT ID routine translates a root aggregate type to an object identifier.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVAGGTYP	Invalid aggregate type.
CDA$_INVBUFLEN	Invalid buffer length.

CDA$CLOSE_FILE CLOSE FILE

Closes the specified compound document file and stream. In the case of an output file, the CLOSE FILE routine writes any buffered data before closing the file and stream.

FORMAT

CDA$CLOSE_FILE *stream-handle ,file-handle*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

stream-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Handle of the stream to be closed. The **stream-handle** argument is the address of an unsigned longword containing this stream handle. This handle is returned by a call to either the OPEN FILE routine or the CREATE FILE routine.

file-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Handle of the file to be closed. The **file-handle** argument is the address of an unsigned longword containing this file handle. This handle is returned by a call to either the OPEN FILE routine or the CREATE FILE routine.

DESCRIPTION

The CLOSE FILE routine closes the specified stream and compound document file. In the case of an output stream, this routine writes out any buffered data before closing the stream. Note that the **stream-handle** and **file-handle** handles are invalid after a call to this routine.

CONDITION VALUES RETURNED

CDA$_NORMAL Normal successful completion.

Any error returned by the memory deallocation routines.

Any error returned by the file routines.

EXAMPLE

```
    .
    .
    .
/* output to a DDIF file */
printf("Writing document...\n");

status = cda$put_document(&root_aggregate_handle, &stream_handle);
if (FAILURE(status)) return(status);

status = cda$close_file(&stream_handle, &file_handle);
if (FAILURE(status)) return(status);

status = cda$delete_root_aggregate(&root_aggregate_handle);
if (FAILURE(status)) return(status);

    .
    .
    .
```

This example illustrates a typical call to the CLOSE FILE routine. The entire document is written to the output file prior to a call to the CLOSE FILE routine. After the file has been closed, the document root aggregate is deleted.

CDA$CLOSE_STREAM CLOSE STREAM

Closes an open compound document stream.

FORMAT

CDA$CLOSE_STREAM *stream-handle*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENT

stream-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Handle of the stream to be closed. The **stream-handle** argument is the address of an unsigned longword containing this stream handle. This handle is returned by a call to either the OPEN STREAM routine or the CREATE STREAM routine.

DESCRIPTION

The CLOSE STREAM routine closes an open compound document stream. In the case of an output stream, the CLOSE STREAM routine writes out any buffered data before closing the stream. Note that the **stream-handle** argument is invalid after a call to this routine.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.

Any error returned by the memory deallocation routines.

CDA$CLOSE_TEXT_FILE CLOSE TEXT FILE

Closes a text file.

FORMAT

CDA$CLOSE_TEXT_FILE *text-file-handle*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENT

text-file-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the text file to be closed. The **text-file-handle** argument is the address of an unsigned longword containing this text file handle. This handle is returned by a call to either the CREATE TEXT FILE routine or the OPEN TEXT FILE routine.

DESCRIPTION

The CLOSE TEXT FILE routine closes a text file. The **text-file-handle** is invalid after a call to this routine.

CONDITION VALUES RETURNED

CDA$_NORMAL Normal successful completion.

Any error returned by the memory deallocation routines.

Any error returned by the file routines.

CDA$CONVERT CONVERT

Lets the user perform document conversion from within an application. This includes beginning, continuing, or discontinuing the conversion of a document.

FORMAT

CDA$CONVERT *function-code ,standard-item-list ,private-item-list ,converter-context*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

function-code

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Symbolic constant that identifies the function to be performed. The **function-code** argument is the address of an unsigned longword that contains this symbolic constant. These symbolic constant values are defined in the file CDA$DEF.SDL. Valid values are as follows:

CDA$_START
Start conversion. This function code must be specified to begin a document conversion.

CDA$_CONTINUE
Continue a conversion that was suspended. This function code may only be specified if a previous call to the CONVERT routine returned the value CDA$_SUSPEND. If CDA$_SUSPEND is returned by a call to the CONVERT routine, either CDA$_CONTINUE or CDA$_STOP must be specified so that resources locked by the conversion are released.

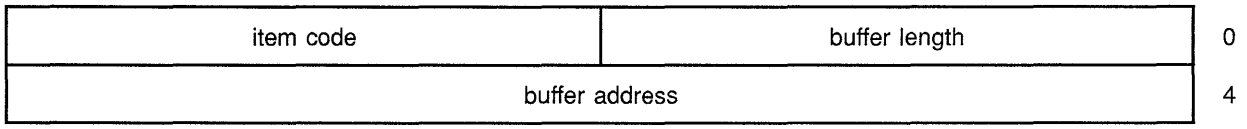
CDA$_STOP
Discontinue a conversion that was suspended. This function code may only be specified if the previous call to the CONVERT routine returned the value CDA$_SUSPEND. If CDA$_SUSPEND is returned by a call to the CONVERT routine, either CDA$_STOP or CDA$_CONTINUE must be specified so that resources locked by the conversion are released.

standard-item-list

VMS usage: **item_list_2**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference, array reference**

An item list that identifies the document source and destination, and can also contain options to control processing. The **standard-item-list** argument is the address of this item list.

Each entry in the item list is a 2-longword structure with the following format:



To terminate the item list, you must specify the final entry or longword as 0. The **standard-item-list** argument is only valid when **function-code** is set to CDA$_START; otherwise, **standard-item-list** is ignored. Valid code values for the items in the **standard-item-list** are as follows:

CDA$_INPUT_FORMAT

The parameter is the address and length of a string that specifies the input document format.

CDA$_INPUT_FRONT_END_PROCEDURE

The parameter is the address of the front end module's main entry point, DDIF$READ_*format*. The term *format* in the entry point name refers to the name of the specific document format that is read by this front end.

The item-list length field for this item must be set to 0. This item enables a caller to provide a front end that is part of the calling application rather than a separate image. If this item code is used, the CDA$_INPUT_ FILE item can be used to pass any information (not necessarily a file specification) to the front end.

CDA$_INPUT_FILE

The parameter is the address and length of the file specification of the input document.

CDA$_INPUT_DEFAULT

The parameter is the address and length of a string that specifies the default input file type. To simplify the porting of applications to other operating systems, the string should consist only of a file type in lowercase characters. If this parameter is omitted, the front end must supply an appropriate backup default file specification.

CDA$_INPUT_PROCEDURE

The parameter is the address of a procedure to provide input. The item-list length field must be set to 0. The input procedure must conform to the requirements for a user *get* routine. The calling sequence for a user *get* routine is defined in the Description section of this routine.

CDA$_INPUT_PROCEDURE_PARM

The parameter is the address of a longword parameter to the input procedure. The item-list length field must be set to 4.

CDA$_INPUT_POSITION_PROCEDURE

The parameter is the address of a procedure that provides position information. The item-list length field must be set to 0. The *get-position* procedure is specified in the description of the OPEN CONVERTER routine.

CDA$_INPUT_ROOT_AGGREGATE

The parameter is the address of a longword root aggregate handle that specifies an in-memory input document. The item-list length field must be set to 4. The in-memory structure, except for the root aggregate itself, is erased by this operation. Note that the root aggregate must specify standard memory allocation.

CDA$_OUTPUT_FORMAT

The parameter is the address and length of a string that specifies the output document format.

CDA$_OUTPUT_BACK_END_PROCEDURE

The parameter is the address of the back end module's main entry point, DDIF$WRITE_*format*. The term *format* in the entry point name refers to the name of the specific document format that is written by this back end.

The item-list length field must be set to 0. This item enables a caller to provide a back end that is part of the calling application rather than a separate image. If this item code is used, the CDA$_OUTPUT_FILE item can be used to pass any information (not necessarily a file specification) to the back end.

CDA$_OUTPUT_FILE

The parameter is the address and length of the file specification of the output document.

CDA$_OUTPUT_DEFAULT

The parameter is the address and length of a string that specifies the default output file type. To simplify the porting of applications to other operating systems, the string should consist only of a file type in lowercase characters. If this parameter is omitted, the back end must supply an appropriate backup default file specification.

CDA$_OUTPUT_PROCEDURE

The parameter is the address of a procedure to receive output. The item-list length field must be set to 0. The output procedure must conform to the requirements for a user *put* routine. The calling sequence for a user *put* routine is defined in the Description section of this routine.

CDA$_OUTPUT_PROCEDURE_PARM

The parameter is the address of a longword parameter to the output procedure. The item-list length field must be set to 4.

CDA$_OUTPUT_PROCEDURE_BUFFER

The parameter is the address and length of the initial output buffer for the output procedure.

CDA$_OUTPUT_ROOT_AGGREGATE
The parameter is the address of a longword root aggregate handle that receives an in-memory output document. The item-list length field must be set to 4. The root aggregate must be empty, and must specify standard memory allocation.

CDA$_OPTIONS_FILE
The parameter is the address and length of the file specification of an options file that contains options to control processing. The default file type is CDA$OPTIONS. Each line of the file specifies a format name, which can contain upper- and lowercase alphabetic characters, digits, dollar signs, and underscores, optionally preceded by spaces and tabs, and terminated by any character other than those listed. Alphabetic case is not significant. The syntax and interpretation of the text that follows the format name is specified by the supplier of the front and back ends for the specified format. Multiple lines that specify the same format are permitted.

private-item-list

VMS usage: **unspecified**
type: **unspecified**
access: **read only**
mechanism: **by reference**

A private item list that is passed directly to the output converter module that is invoked. The **private-item-list** argument is the address of this private item list. The specification of this item list is the responsibility of the particular back end. Its purpose is to provide for direct two-way communication between the caller of the CONVERT routine and the back end.

converter-context

VMS usage: **context**
type: **longword (unsigned)**
access: **read only or write only**
mechanism: **by reference**

If **function-code** is set to CDA$_START, this argument receives a value that must be specified as the **converter-context** parameter when the CONVERT routine is called with CDA$_CONTINUE or CDA$_STOP as the function code. The **converter-context** argument is the address of an unsigned longword that either receives or specifies the converter context. This value is invalidated when the CONVERT routine returns a status other than CDA$_SUSPEND.

DESCRIPTION

The CONVERT routine lets you perform document conversion from within an application. This includes beginning, continuing, or discontinuing the conversion of a document.

To specify the input and output information, and any processing options files, you should construct an item list with the appropriate fields as specified in the description of the **standard-item-list** argument. Note that the **standard-item-list** argument is only valid when **function-code** is set to CDA$_START. The following restrictions apply when you are constructing the **standard-item-list**:

- Either the CDA$_INPUT_FORMAT item or the CDA$_INPUT_FRONT_END_PROCEDURE item, but not both, can be specified once in the item list. If neither is specified, the default input format is DDIF.
- The CDA$_INPUT_FILE item, the CDA$_INPUT_PROCEDURE item, or the CDA$_INPUT_ROOT_AGGREGATE item must be specified once in the item list. If the CDA$_INPUT_PROCEDURE item is specified, the CDA$_INPUT_PROCEDURE_PARM item can also be specified once.
- Either the CDA$_OUTPUT_FORMAT item or the CDA$_OUTPUT_BACK_END_PROCEDURE item, but not both, can be specified once in the item list. If neither is specified, the default output format is DDIF.
- The CDA$_OUTPUT_FILE item, the CDA$_OUTPUT_PROCEDURE item, or the CDA$_OUTPUT_ROOT_AGGREGATE item must be specified once in the item list. If the CDA$_OUTPUT_PROCEDURE item is specified, the CDA$_OUTPUT_PROCEDURE_PARM item and the CDA$_OUTPUT_PROCEDURE_BUFFER item can each be specified once.
- The CDA$_OPTIONS_FILE item can only be specified once in the item list.

Call Format for User *Get* Routines

The **get-rtn** and **get-prm** arguments are used to invoke a user stream *get* routine, and to supply an argument to that routine. The call format for this user routine is as follows:

get-rtn get-prm ,num-bytes ,buf-adr

get-prm
VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**
User context argument. The **get-prm** argument contains the value of the parameter to be passed to the user *get* routine.

num-bytes
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives the number of bytes contained in the buffer. The **num-bytes** argument is the address of an unsigned longword that receives this number. The number of bytes is zero if and only if the stream does not contain any more data.

buf-adr
VMS usage: **address**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the address of the buffer. The **buf-adr** argument is the address of an unsigned longword that receives the buffer address.

Call Format for User *Put* Routines

The **put-rtn** and **put-prm** arguments are used to invoke a user stream *put* routine, and to supply an argument to that routine. The call format for this user routine is as follows:

put-rtn put-prm ,num-bytes ,buf-adr ,next-buf-len ,next-buf-adr

put-prm
VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**
User context argument. The **put-prm** argument is the value of the parameter to be passed to the user *put* routine.

num-bytes
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**
Number of bytes contained in the buffer. The **num-bytes** argument is the address of an unsigned longword that contains this value.

buf-adr
VMS usage: **vector_byte_unsigned**
type: **byte (unsigned)**
access: **read only**
mechanism: **by reference, array reference**
Address of the buffer. The **buf-adr** argument is the address of an array of unsigned bytes.

next-buf-len
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives the length of the buffer specified by **next-buf-adr**. The **next-buf-len** argument is the address of an unsigned longword that receives this length.

next-buf-adr
VMS usage: **address**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives the address of a buffer that will receive further output data. The **next-buf-adr** argument is the address of an unsigned longword that receives this address. **Next-buf-adr** may simply be the current buffer, or a different buffer.

Each of these user routines must return a completion status. The VMS convention for completion codes is followed: if the low bit of the return value is clear, an error has occurred and the caller returns control to its caller; if the low bit of the return value is set, the caller continues execution.

Front End Module

The DDIF$READ_*format* routine executes a compound document conversion from a specified input format to the CDA in-memory format. The term *format* in the entry point name refers to the name of the compound document format that is read by this front end.

Each front end must supply the DDIF$READ_*format* entry point. As described in the CDA$_INPUT_PROCEDURE item list discussion, the procedure address for the front end may be specified as a parameter to the CONVERT routine. Otherwise, on VMS systems, the front end must be a universal symbol in a shareable image named SYS$SHARE:DDIF$READ_*format*.EXE.

The call format for a front end is as follows:

DDIF$READ_*format* standard-item-list
,converter-context ,front-end-context
,get-aggregate-procedure ,get-position-procedure
,close-procedure

standard-item-list
VMS usage: **item_list_2**
type: **record**
access: **read only**
mechanism: **by reference, array reference**
An item list that identifies the document source and can also contain options to control processing. The **standard-item-list** argument is the address of this item list.

Each entry in the item list is a 2-longword structure with the following format:

item code	buffer length	0
buffer address		4

To terminate the item code you must specify the final entry or longword as 0. The **standard-item-list** argument is only valid when the **function-code** argument is set to CDA$_START; otherwise, **standard-item-list** is ignored. Valid code values for the items in the **standard-item-list** are as follows:

CDA$_INPUT_FILE
The parameter is the address and length of the file specification of the input document.

CDA$_INPUT_DEFAULT
The parameter is the address and length of a string that specifies the default input file type. To simplify the porting of applications, the string should consist of only a file type in lowercase characters. If this parameter is omitted, a front end must supply an appropriate backup default file specification.

CDA$_INPUT_PROCEDURE
The parameter is the address of a procedure to provide input. The item-list length field must be set to 0. The input procedure must conform to the requirements for a user *get* routine. The calling sequence for a user *get* routine is defined in the Description section of this routine.

CDA$_INPUT_PROCEDURE_PARM
The parameter is the address of a longword parameter to the input procedure. The item-list length field must be set to 4.

CDA$_PROCESSING_OPTION
The parameter is the address and length of a string that contains an option to control processing. The format name and leading spaces and tabs have been removed from the string. This item code may occur more than once in the item list.

Either the CDA$_INPUT_FILE item or the CDA$_INPUT_PROCEDURE item, but not both, must occur once in the item list. If the CDA$_INPUT_PROCEDURE item occurs, then a single value for CDA$_INPUT_PROCEDURE_PARM can also be specified.

converter-context
VMS usage: **context**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**
Converter context required to call the OPEN CONVERTER routine. The **converter-context** argument is the address of an unsigned longword that contains this context.

front-end-context
VMS usage: **context**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives a front-end-defined value that identifies this particular front end. The **front-end-context** argument is the address of an unsigned longword that receives this context. This value is returned to the **get-aggregate-procedure** and the **close-procedure** arguments described below. All writable memory used by the input converter module must be allocated from dynamic memory and located by reference to this value.

get-aggregate-procedure
VMS usage: **procedure**
type: **procedure entry mask**
access: **write only**
mechanism: **by reference**
Receives the address of the *get-aggregate* routine. The **get-aggregate-procedure** argument receives the address of this procedure entry mask. The calling sequence for the *get-aggregate* routine is described in the Description section of this routine.

get-position-procedure
VMS usage: **procedure**
type: **procedure entry mask**
access: **write only**
mechanism: **by reference**
Receives the address of the *get-position* routine. The **get-position-procedure** argument receives the address of this procedure entry mask. The calling sequence for the *get-position* routine is described in the Description section of this routine.

close-procedure
VMS usage: **procedure**
type: **procedure entry mask**
access: **write only**
mechanism: **by reference**
Receives the address of the *close* routine. The **close-procedure** argument receives the address of this procedure entry mask. The calling sequence for the *close* routine is described in the Description section of this routine.

The possible status codes that DDIF$READ_*format* can return are either CDA$_NORMAL, or any input converter-specific error conditions.

Call Format for *Get-Aggregate* Procedure

The *get-aggregate* procedure returns the handle and type of the next aggregate in the document to the converter kernel. For more information on the function of a *get-aggregate* procedure, see Chapter 5.

The call format for the *get-aggregate* procedure is as follows:

get-aggregate-procedure front-end-context ,aggregate-handle
,aggregate-type

front-end-context
VMS usage: **context**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**
Context returned from DDIF$READ_*format*. The **front-end-context** argument is the address of an unsigned longword that contains this context.

aggregate-handle
VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives the handle of the created and populated aggregate. The **aggregate-handle** argument is the address of an unsigned longword that receives this aggregate handle. This handle must be used in all subsequent operations on that aggregate.

aggregate-type
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives the aggregate type. The **aggregate-type** argument is the address of an unsigned longword that receives this aggregate type. If the aggregate is of type DDIF$_EOS (end of segment), **aggregate-handle** is 0.

The possible status codes that a *get-aggregate* procedure can return are as follows:

CDA$_NORMAL	Normal successful completion.
CDA$_ENDOFDOC	End of document.

A *get-aggregate* procedure can also return any front end-specific error conditions. Note that the *get-aggregate* procedure must return the status CDA$_ENDOFDOC when the document has been completely transferred.

Call Format for a User *Get-Position* Procedure

The user *get-position* routine returns the current position in and total size of the current data stream. The call format for this routine is as follows:

get-position-procedure front-end-handle ,stream-position
,stream-size

front-end-handle
VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**
Identifier of the front end performing document processing. The **front-end-handle** argument is the address of an unsigned longword that contains this handle. The front end handle is returned by DDIF$READ_*format*.

stream-position
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives the current position (in bytes) as measured from the start of the input stream being processed. The **stream-position** argument is the address of an unsigned longword that receives this position.

stream-size
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives the total size (in bytes) of the input stream being processed. The **stream-size** argument is the address of an unsigned longword that receives this size.

Call Format for a User *Close* Procedure

The *close* procedure terminates the operation of a front end by closing all open files and releasing all dynamic memory and other resources that have been allocated by the particular front end identified by **front-end-context**.

The call format for a *close* procedure is as follows:

close-procedure front-end-context

front-end-context
VMS usage: **context**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**
Context returned from DDIF$READ_*format*. The **front-end-context** argument is the address of an unsigned longword that contains this context.

The possible status codes that a *close* procedure can return are either CDA$_NORMAL, or any front end-specific error conditions.

Back End

The DDIF$WRITE_*format* procedure executes a document conversion from the DDIF in-memory format to a specified output format. The term *format* in the entry point name refers to the name of the document format that is written by this back end.

Each back end must supply the DDIF$WRITE_*format* entry point. As described in the CDA$_OUTPUT_PROCEDURE item list discussion, the procedure address for the back end may be specified as a parameter to the CONVERT routine. Otherwise, on VMS systems, the back end must be a universal symbol in a shareable image named SYS$SHARE:DDIF$WRITE_*format*.EXE.

The call format for a back end is as follows:

DDIF$WRITE_*format* function-code
,standard-item-list ,private-item-list
,front-end-handle ,back-end-context

function-code
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Symbolic constant that identifies the function to be performed. The **function-code** argument is the address of an unsigned longword that contains this symbolic constant. These constant values are defined in module CDA$DEF.SDL. Valid values are as follows:

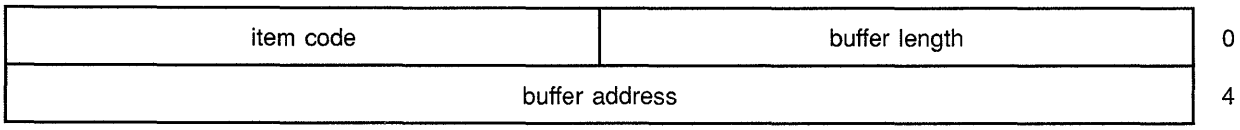
CDA$_START
Start conversion. This function code must be specified to begin a document conversion.

CDA$_CONTINUE
Continue a conversion that was suspended. This function code can only be specified if a previous call to DDIF$WRITE_*format* returned the value CDA$_SUSPEND. If CDA$_SUSPEND is returned by a call to the back end, either CDA$_CONTINUE or CDA$_STOP must be specified so that resources locked by the conversion can be released.

CDA$_STOP
Discontinue a conversion that was suspended. This function code can only be specified if the previous call to the back end returned the value CDA$_SUSPEND. If CDA$_SUSPEND is returned by a call to the back end, either CDA$_STOP or CDA$_CONTINUE must be specified so that resources locked by the conversion can be released.

standard-item-list
VMS usage: **item_list_2**
type: **record**
access: **read only**
mechanism: **by reference, array reference**
An item list that identifies the document destination and can also contain options to control processing. The **standard-item-list** argument is the address of this item list.

Each entry in the item list is a 2-longword structure with the following format:



To terminate the item list, you must specify the final entry or longword as 0. The **standard-item-list** argument is only valid when the **function-code** argument is set to CDA$_START; otherwise, **standard-item-list** is ignored. Valid code values for the items in the **standard-item-list** are as follows:

CDA$_OUTPUT_FILE
The parameter is the address and length of the file specification of the output document.

CDA$_OUTPUT_DEFAULT
The parameter is the address and length of the default file specification of the output document. If this parameter is omitted, the back end must supply an appropriate backup default file specification.

CDA$_OUTPUT_PROCEDURE
The parameter is the address of a procedure to receive output. The item-list length field must be set to 0. The output procedure must conform to the requirements for a user *put* routine. The calling sequence for a user *put* routine is defined in the Description section of this routine.

CDA$_OUTPUT_PROCEDURE_PARM
The parameter is the address of a longword parameter to the output procedure. The item-list length field must be set to 4.

CDA$_OUTPUT_PROCEDURE_BUFFER
The parameter is the address and length of the initial output buffer for the output procedure.

CDA$_PROCESSING_OPTION
The parameter is the address and length of a string that contains options to control processing. The format name and leading spaces and tabs have been removed from the string. This item code can occur more than once in the item list.

Either the CDA$_OUTPUT_FILE item or the CDA$_OUTPUT_PROCEDURE item, but not both, must occur once in the item list. If you specify the CDA$_OUTPUT_PROCEDURE item, then you can also specify a single value for both the CDA$_OUTPUT_PROCEDURE_PARM item and the CDA$_OUTPUT_PROCEDURE_BUFFER item.

private-item-list
VMS usage: **unspecified**
type: **unspecified**
access: **read only**
mechanism: **by reference**
A private item list that is passed directly to the back end that is invoked. The **private-item-list** argument is the address of this private item list. The specification of this item list is the responsibility of the back end. Its purpose is to provide for direct 2-way communication between the caller of the CONVERT routine and the back end.

front-end-handle
VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**
Identifier of the front end that processes the document content. The **front-end-handle** argument is the address of an unsigned longword that contains this front end handle. This identifier is passed to the CONVERT DOCUMENT routine or the CONVERT AGGREGATE routine.

back-end-context
VMS usage: **context**
type: **longword (unsigned)**
access: **read only or write only**
mechanism: **by reference**
When **function-code** is set to CDA$_START, this argument receives a value defined by the back end that identifies this particular instance of the back end. The **back-end-context** argument is the address of an unsigned

longword that either receives or specifies the converter context. This value will be returned to DDIF$WRITE_*format* for the functions CDA$_CONTINUE and CDA$_STOP. If a back end returns CDA$_SUSPEND, all writable memory used by the back end must be allocated from dynamic memory and located by reference to this value.

The possible status codes that DDIF$WRITE_*format* can return are as follows:

CDA$_NORMAL	Normal successful completion.
CDA$_SUSPEND	Converter is suspended.
CDA$_INVFUNCOD	Invalid function code.
CDA$_INVITMLST	Invalid item list.
CDA$_UNSUPFMT	Unsupported document format.

Back ends can also return any error returned by the specific back end, or any error returned by the specific front end.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_SUSPEND	Converter is suspended.
CDA$_UNSUPFMT	Unsupported document format.

Any error conditions returned by the specific front end.

Any error conditions returned by the specific back end.

EXAMPLE

```
/* TEXT_CONV -  test callable converter interface for DDIF
 *  and TEXT converters.
 */

#ifdef vms
#include <cda$def.h>
#include <cda$msg.h>
#include <fab.h>
#include <rab.h>
#include <rmsdef.h>
#else
#include <cda_def.h>
#include <cda_msg.h>
#include <sys/file.h>
#endif

#define FAILURE(x)     (((x) & 1) == 0)

#define text_ubf_size 2048

#ifdef vms
struct FAB      text_fab;
struct RAB      text_rab;
#else
struct urab     text_rab;
#endif
unsigned char   text_ubf[text_ubf_size];

static unsigned char ddif_format[] = "DDIF";
static unsigned long ddif_format_length = sizeof(ddif_format) - 1;
```

```
static unsigned char text_format[] = "TEXT";
static unsigned long text_format_length = sizeof(text_format) - 1;

static unsigned char text_file[] = "text";
static unsigned long text_file_length = sizeof(text_file) - 1;

static unsigned char text_default[] = ".txt";
static unsigned long text_default_length = sizeof(text_default) - 1;

static unsigned char ddif_file[] = "output";
static unsigned long ddif_file_length = sizeof(ddif_file) - 1;

static unsigned char ddif_default[] = ".ddif";
static unsigned long ddif_default_length = sizeof(ddif_default) - 1;

unsigned long input_text_procedure(get_prm, num_bytes, buf_adr)
#ifdef vms
struct RAB      *get_prm;
#else
struct urab     *get_prm;
#endif
unsigned long   *num_bytes;
unsigned char   **buf_adr;
{
unsigned long   status;

#ifdef vms
        status = sys$get(get_prm);
        if (FAILURE(status))
            {
            if (status == RMS$_EOF)
                status = CDA$_ENDOFDOC;
            return status;
            }
        *num_bytes = get_prm->rab$w_rsz;
        *buf_adr = get_prm->rab$l_rbf;
        return status;
#else
unsigned long buffer_length;

        status = fgets(get_prm->fil_buffer, get_prm->fil_buflen, get_prm->fs);
        if (status == NULL)
        {
                *num_bytes = 0;
                return CDA$_ENDOFDOC;
        }
        buffer_length = strlen(get_prm->fil_buffer);
/*      if ((get_prm->fil_buffer)[buffer_length-1] == '\n')
            *num_bytes = buffer_length - 1;
        else */
            *num_bytes = buffer_length;
        *buf_adr = get_prm->fil_buffer;
        return CDA$_NORMAL;
#endif
}
main()
{
unsigned long status;
unsigned long text_parameter;
struct item_list standard_item_list[15];
unsigned long integer_value;
unsigned long index;
unsigned char text_filename[8];
        printf ("Starting TEXT to DDIF procedure conversion\n");
```

```
#ifdef vms
        /* Open input text file */
        text_fab = cc$rms_fab;
        text_rab = cc$rms_rab;
        text_fab.fab$l_fna = text_file;
        text_fab.fab$b_fns = text_file_length;
        text_fab.fab$l_fop = FAB$M_SQO;
        text_fab.fab$b_rfm = FAB$C_VAR;
        text_fab.fab$l_dna = text_default;
        text_fab.fab$b_dns = text_default_length;
        text_rab.rab$l_fab = &text_fab;
        text_rab.rab$l_rop = RAB$M_LOC | RAB$M_RAH;
        text_rab.rab$l_ubf = text_ubf;
        text_rab.rab$w_usz = text_ubf_size;

        status = sys$open(&text_fab);
        if (FAILURE(status)) return status;
        status = sys$connect(&text_rab);
        if (FAILURE(status))
            {
            sys$close(&text_fab);
            return status;
            }
#else
        strcpy(text_filename, text_file);
        strcat(text_filename, text_default);
        text_filename[text_file_length + text_default_length] = 0;
        text_rab.fil_buffer = &text_ubf;
        text_rab.fil_buffer = &text_ubf;
        text_rab.fil_buflen = text_ubf_size;
        text_rab.fs = fopen(text_filename, "r");
        if (text_rab.fs == NULL) return CDA$_OPENFAIL;
#endif

        /* Setup for conversion */
        text_parameter = (unsigned long) &text_rab;

        integer_value = CDA$_START;

        /* Input conversion parameters */
        index = 0;
        standard_item_list[index].cda$w_item_length = text_format_length;
        standard_item_list[index].cda$w_item_code = CDA$_INPUT_FORMAT;
        standard_item_list[index].cda$a_item_address = (char *) &text_format;
        index += 1;
        standard_item_list[index].cda$w_item_length = 0;
        standard_item_list[index].cda$w_item_code = CDA$_INPUT_PROCEDURE;
        standard_item_list[index].cda$a_item_address = (char *)
                        &input_text_procedure;
        index += 1;
        standard_item_list[index].cda$w_item_length = 4;
        standard_item_list[index].cda$w_item_code = CDA$_INPUT_PROCEDURE_PARM;
        standard_item_list[index].cda$a_item_address = (char *)
                        &text_parameter;
        index += 1;
```

```
        /* Output conversion parameters */
        standard_item_list[index].cda$w_item_length = ddif_format_length;
        standard_item_list[index].cda$w_item_code = CDA$_OUTPUT_FORMAT;
        standard_item_list[index].cda$a_item_address = (char *) &ddif_format;
        index += 1;
        standard_item_list[index].cda$w_item_length = ddif_file_length;
        standard_item_list[index].cda$w_item_code = CDA$_OUTPUT_FILE;
        standard_item_list[index].cda$a_item_address = (char *) &ddif_file;
        index += 1;
        standard_item_list[index].cda$w_item_length = ddif_default_length;
        standard_item_list[index].cda$w_item_code = CDA$_OUTPUT_DEFAULT;
        standard_item_list[index].cda$a_item_address = (char *) &ddif_default;
        index += 1;
        standard_item_list[index].cda$w_item_length = 0;
        standard_item_list[index].cda$w_item_code = 0;

        /* Perform the conversion */
        status = cda$convert(&integer_value, standard_item_list, 0,
                             &integer_value);
 if (FAILURE(status))
  return (status);

#ifdef vms
        /* Close the input file */
        status = sys$close(&text_fab);
        if (FAILURE(status)) return status;
#else
        fclose(text_rab.fs);
#endif

        printf ("Completed TEXT to DDIF procedure conversion\n");
}
```

This example illustrates the use of the CONVERT routine to invoke the DDIF and Text converters. To compile this program, you must use the following DCL commands:

```
$ CC /INCLUDE=DDIF$LIB_SRC: -
_$ /OPTIMIZE=NODISJOINT -
_$ /OBJECT = TEXT_CONV -
_$ /NOLIST -
_$ TEXT_CONVERTER.C

$ LINK /EXE=TEXT_CONVERTER -
_$ /NOMAP -
_$ TEXT_CONVERTER,SYS$INPUT:/OPTION
DDIF$LIB_OBJ:CDA$ACCESS/SHARE
SYS$SHARE:VAXCRTL/SHARE

$ RUN TEXT_CONVERTER
```

CDA$CONVERT_AGGREGATE CONVERT AGGREGATE

Reads the next aggregate from a specified front end.

FORMAT

CDA$CONVERT_AGGREGATE
root-aggregate-handle
,front-end-handle ,aggregate-handle
,aggregate-type

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate associated with the aggregate to be read. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This handle is returned by a call to the CREATE ROOT AGGREGATE routine.

When reading aggregates using this routine, you must use the same value for **root-aggregate-handle** consistently to read all the aggregates in the compound document. Once you have read all of the aggregates, you cannot specify the same **root-aggregate-handle** again when calling this routine.

front-end-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the front end that reads the aggregate. The **front-end-handle** argument is the address of an unsigned longword that contains this front end handle. This handle is either returned by a call to the OPEN CONVERTER routine or is passed as a parameter to the DDIF$WRITE_*format* entry point in the back end.

aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the handle of the aggregate just read. The **aggregate-handle** argument is the address of an unsigned longword that receives this aggregate handle. This handle must be used in all subsequent operations on that aggregate.

aggregate-type

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the aggregate type. The **aggregate-type** argument is the address of an unsigned longword that receives this aggregate type. If the aggregate type is DDIF$_EOS (end of segment), then the value of **aggregate-handle** is 0.

DESCRIPTION

The CONVERT AGGREGATE routine lets you call through from a back end to read the next aggregate from the specified front end. This routine should only be invoked by a back end.

The aggregate type returned is one of the following:

Aggregate Type	Meaning
DDIF$_DSC	Document descriptor
DDIF$_DHD	Document header
DDIF$_SEG	Document segment
DDIF$_TXT	Text content
DDIF$_GTX	General text content
DDIF$_HRD	Hard directive
DDIF$_SFT	Soft directive
DDIF$_HRV	Hard value directive
DDIF$_SFV	Soft value directive
DDIF$_BEZ	Bézier curve content
DDIF$_LIN	Polyline content
DDIF$_ARC	Arc content
DDIF$_FAS	Fill area set content
DDIF$_IMG	Image content
DDIF$_CRF	Content reference
DDIF$_EXT	External content
DDIF$_PVT	Private content
DDIF$_GLY	Layout galley
DDIF$_EOS	End of segment

Note that the returned aggregate is not part of a sequence.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_ENDOFDOC	End of document.
CDA$_INVDOC	Invalid document content.

Any error returned by the memory allocation routines.

Any error returned by the file routines.

CDA$CONVERT_DOCUMENT CONVERT DOCUMENT

Reads an entire document from a specified front end.

FORMAT

CDA$CONVERT_DOCUMENT *root-aggregate-handle ,front-end-handle*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate associated with the document being read. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This root aggregate handle is returned by a call to the CREATE ROOT AGGREGATE routine.

Once you read an entire document, you cannot call the CONVERT DOCUMENT routine specifying the same root aggregate handle again. That is, you can only read a document associated with a particular root aggregate once.

front-end-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the front end that reads the document. The **front-end-handle** argument is the address of an unsigned longword that contains this front end handle. This handle is either returned by a call to the OPEN CONVERTER routine, or is passed as a parameter to the DDIF$WRITE_*format* entry point in the back end.

DESCRIPTION

The CONVERT DOCUMENT routine lets you call through from a back end to read an entire document from the specified front end. This routine should only be invoked by a back end. On return from this routine, the entire document is present in aggregates linked from the document root aggregate.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVAGGTYP	Invalid aggregate type.
CDA$_INVDOC	Invalid document content.

Any error returned by the memory allocation routines.

Any error returned by the file routines.

CDA$CONVERT_POSITION CONVERT POSITION

Returns the current position in and total size of the input stream being processed.

FORMAT

CDA$CONVERT_POSITION
front-end-handle ,stream-position ,stream-size

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

front-end-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the front end that is processing the input stream. The **front-end-handle** argument is the address of an unsigned longword that contains this identifier. The front end handle is either returned by a call to the OPEN CONVERTER routine, or is passed as a parameter to DDIF$WRITE*_format*.

stream-position

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the current position (in bytes) as measured from the start of the input stream being processed. The **stream-position** argument is the address of an unsigned longword that receives this position.

stream-size

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the total size (in bytes) of the input stream being processed. The **stream-size** argument is the address of an unsigned longword that receives this size.

DESCRIPTION

The CONVERT POSITION routine returns the current position in and total size of a document being processed by the CONVERT AGGREGATE routine.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.

Any condition value returned by the front end *get-position* procedure.

CDA$COPY_AGGREGATE COPY AGGREGATE

Creates a copy of an aggregate. If the specified aggregate is part of a sequence, only the aggregate specified, rather than the entire sequence, is copied.

FORMAT

CDA$COPY_AGGREGATE *root-aggregate-handle*
,input-aggregate-handle
,output-aggregate-handle

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate with which the copied aggregate is associated. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. The new copy of the aggregate becomes part of the document identified by this root aggregate handle. This root aggregate handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

input-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the aggregate to be copied. The **input-aggregate-handle** argument is the address of an unsigned longword that contains this aggregate handle.

output-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the handle of the new copy of the specified aggregate. The **output-aggregate-handle** argument is the address of an unsigned longword that receives the handle of the new aggregate. This new aggregate handle must be used in all subsequent operations on that aggregate.

CDA$COPY_AGGREGATE

DESCRIPTION

The COPY AGGREGATE routine makes a copy of the specified aggregate. This copy becomes part of the document identified by the specified root aggregate handle argument, and it is assigned a unique aggregate identifier, specified by the output aggregate handle argument.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVAGGTYP	Invalid aggregate type.

CDA$CREATE_AGGREGATE CREATE AGGREGATE

Creates a new aggregate that contains empty items. Once this aggregate is created, it can be populated using the STORE ITEM routine.

FORMAT

CDA$CREATE_AGGREGATE *root-aggregate-handle*
,aggregate-type
,aggregate-handle

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate with which the newly created aggregate is associated. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

aggregate-type

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The type of aggregate to be created, expressed as a symbolic constant. The **aggregate-type** argument is the address of an unsigned longword that specifies this aggregate type. The aggregate type symbolic constants are defined in the module DDIF$DEF.SDL.

aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the identifier of the newly created aggregate. The **aggregate-handle** argument is the address of an unsigned longword that receives this aggregate handle. This handle must be used in all subsequent operations on that aggregate.

CDA$CREATE_AGGREGATE

DESCRIPTION

The CREATE AGGREGATE routine creates a new aggregate that contains empty items. Once this aggregate is created, it can be populated using the STORE ITEM routine. The created aggregate is part of the document specified by the root aggregate handle.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVAGGTYP	Invalid aggregate type.

Any error returned by the memory allocation routines.

CDA$CREATE_FILE CREATE FILE

Creates a new compound document file for output. An output stream is also created.

FORMAT

CDA$CREATE_FILE *file-spec-len ,file-spec ,default-file-spec-len ,default-file-spec ,alloc-rtn ,dealloc-rtn ,alloc-dealloc-prm ,root-aggregate-handle ,result-file-spec-len ,result-file-spec ,result-file-ret-len ,stream-handle ,file-handle*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

file-spec-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The length (in bytes) of the string specified by the **file-spec** parameter. The **file-spec-len** argument is the address of an unsigned longword that contains this file specification length.

file-spec

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by reference**

The file specification. The **file-spec** argument is the address of a character string that contains this file specification.

default-file-spec-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The length (in bytes) of the buffer specified by **default-file-spec**. The **default-file-spec-len** argument is the address of an unsigned longword that contains this buffer length. If you specify a value of 0 for both the **default-file-spec-len** and **default-file-spec** arguments, a default file specification of ".DDIF" is used.

default-file-spec

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by reference**

The default file specification. The **default-file-spec** argument is the address of a character string that contains the default file specification. In order to simplify the porting of applications, the character string should consist of only a file type in lowercase characters. If you specify an address of 0 for both the **default-file-spec-len** and **default-file-spec** arguments, a default file specification of ".DDIF" is used.

alloc-rtn

VMS usage: **procedure**
type: **procedure entry mask**
access: **call after stack unwind**
mechanism: **by reference, procedure reference**

Address of a memory allocation routine. The **alloc-rtn** argument is the address of a procedure entry mask for this allocation routine. The calling sequence for an allocation routine is defined in the Description section of this routine. On VMS systems, if you specify 0 for this argument, LIB$GET_VM is used as the memory allocation routine.

dealloc-rtn

VMS usage: **procedure**
type: **procedure entry mask**
access: **call after stack unwind**
mechanism: **by reference, procedure reference**

Address of a memory deallocation routine. The **dealloc-rtn** argument is the address of a procedure entry mask for this deallocation routine. The calling sequence for a deallocation routine is defined in the Description section of this routine. On VMS systems, if you specify 0 for this argument, LIB$FREE_VM is used as the memory deallocation routine.

alloc-dealloc-prm

VMS usage: **context**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

User context to be passed to the memory allocation and deallocation routines. The **alloc-dealloc-prm** argument contains the value of this user context. If the VMS system default memory allocation or deallocation routine is used, this value must be a zone identifier or 0 for the default zone.

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate associated with the newly created compound document. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle.

This handle must be used in all subsequent operations on that root aggregate.

The **root-aggregate-handle** argument is used to specify the file type of the newly created document using the aggregate type DDIF$_DDF.

result-file-spec-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Length (in bytes) of the buffer specified by **result-file-spec**. The **result-file-spec-len** argument is the address of an unsigned longword containing this length. If you specify 0 for this parameter, the length of the resultant file specification is not returned.

result-file-spec

VMS usage: **char_string**
type: **character string**
access: **write only**
mechanism: **by reference**

Receives the resultant file specification. The **result-file-spec** argument is the address of a character string that receives this file specification. If you specify 0 for this parameter, the resultant file specification is not returned. This file specification is the result of a VMS RMS $CREATE operation.

result-file-ret-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the actual length (in bytes) of the resultant file specification. The **result-file-ret-len** argument is the address of an unsigned longword that receives the actual length of the resultant file specification. If you specify 0 for this parameter, the actual length of the resultant file specification is not returned.

stream-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the handle of the newly created compound document stream. The **stream-handle** argument is the address of an unsigned longword that receives this stream handle. This handle must be used in all subsequent operations on that stream.

file-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the handle of the newly created compound document file. The **file-handle** argument is the address of an unsigned longword that receives

this file handle. This handle must be used in all subsequent operations on that file.

DESCRIPTION

The CREATE FILE routine creates a new compound document file for output and also creates an output stream. Note that you must have created a document root aggregate (by a call to the CREATE ROOT AGGREGATE routine) prior to calling this routine. The handle of this document root aggregate must be passed to the CREATE FILE routine, and must also be used in all subsequent operations on that root aggregate.

Call Format for User Allocation/Deallocation Routines

The **alloc-rtn**, **dealloc-rtn**, and **alloc-dealloc-prm** arguments are used to invoke a user routine that performs memory allocation or deallocation, and to supply an argument to that routine. The call format for each of these user routines is as follows:

user-rtn num-bytes ,base-adr ,alloc-dealloc-prm

num-bytes
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**
The number of bytes to allocate or free. The **num-bytes** argument is the address of an unsigned longword that contains this number of bytes. The value of **num-bytes** must be greater than zero.

base-adr
VMS usage: **address**
type: **longword (unsigned)**
access: **read only or write only**
mechanism: **by reference**
Virtual address of the first byte of memory allocated or freed. The **base-adr** argument is the address of an unsigned longword containing this base address. (This argument is write-only for a *get* routine, and read-only for a *free* routine.)

alloc-dealloc-prm
VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**
User context argument. The **alloc-dealloc-prm** argument contains the value of the parameter to be passed to the user routine.

Each of these user routines must return a completion status. The VMS convention for completion codes is followed: if the low bit of the return value is clear, an error has occurred and the caller returns control to its caller; if the low bit of the return value is set, the caller continues execution.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVAGGTYP	Invalid aggregate type.

Any error returned by the memory allocation routines.

Any error returned by the file routines.

EXAMPLE

```
    .
    .
    .
/* set up file for DDIF file */
spec_length = 12;
result_length = sizeof(result_buffer);
status = cda$create_file(&spec_length, "example.ddif", 0, 0,
                         0, 0, 0,
                         &root_aggregate_handle, &result_length,
                         &result_buffer[0], &result_length,
                         &stream_handle, &file_handle);
if (FAILURE(status)) return(status);

    .
    .
    .
```

This example illustrates a typical call to the CREATE FILE routine. The length of the file specification is specified by the **spec_length** parameter, and the file specification itself is "example.ddif". This call does not specify a default file specification length or a default file specification; this combination defaults to a default file specification of ".ddif". The system memory allocation and deallocation routines are passed as a zero value, meaning that the default system memory routines are used. The default system memory routines are LIB$GET_VM and LIB$FREE_VM. The root aggregate handle specifies the root aggregate of the document. This root aggregate must exist prior to a call to this routine.

The **result_length**, **result_buffer**, and **result_length** arguments contain information about the actual resultant file specification of the created file. The **stream_handle** and **file_handle** arguments receive the identifiers of the newly created stream and file.

CDA$CREATE_ROOT_AGGREGATE CREATE ROOT AGGREGATE

Creates a document root aggregate.

FORMAT

CDA$CREATE_ROOT_AGGREGATE
alloc-rtn ,dealloc-rtn ,alloc-dealloc-prm ,processing-options ,aggregate-type ,root-aggregate-handle

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

alloc-rtn

VMS usage: **procedure**
type: **procedure entry mask**
access: **call after stack unwind**
mechanism: **by reference, procedure reference**

Address of a memory allocation routine. The **alloc-rtn** argument is the address of a procedure entry mask for this allocation routine. The calling sequence for an allocation routine is defined in the Description section of this routine. On VMS systems, if you specify 0 for this argument, LIB$GET_VM is used as the memory allocation routine.

dealloc-rtn

VMS usage: **procedure**
type: **procedure entry mask**
access: **call after stack unwind**
mechanism: **by reference, procedure reference**

Address of a memory deallocation routine. The **dealloc-rtn** argument is the address of a procedure entry mask for this deallocation routine. The calling sequence for a deallocation routine is defined in the Description section of this routine. On VMS systems, if you specify 0 for this argument, LIB$FREE_VM is used as the memory deallocation routine.

alloc-dealloc-prm

VMS usage: **context**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

User context to be passed to the memory allocation and deallocation routines. The **alloc-dealloc-prm** argument is the value of this user context. If the VMS system default memory allocation or deallocation routine is used, this value must be a zone identifier or 0 for the default zone.

processing-options

VMS usage: **item_list_2**
type: **record**
access: **read only**
mechanism: **by reference, array reference**

An item list containing options to control input processing. The **processing-options** argument is the address of this item list. Each entry in the item list is a 2-longword structure. To terminate the item list you must specify the final entry or longword as 0. Valid item codes are as follows:

DDIF$_INHERIT_ATTRIBUTES	If a style guide is specified in the document header, definitions in the style guide are appended to the definitions present on the root segment, provided they are not hidden by definitions in the document.
DDIF$_RETAIN_DEFINITIONS	Segment definitions that enable the operation of CDA$FIND_DEFINITION are retained. This item code is required only if neither DDIF$_INHERIT_ATTRIBUTES nor DDIF$_EVALUATE_CONTENT is specified.
DDIF$_EVALUATE_CONTENT	If a content reference is external, the content is fetched from the external document provided it is either remote content or copy content that is not present in the document.
DDIF$_DISCARD_I_SEGMENTS	Segments of the image ($I) content category, and any nested segments, are discarded.
DDIF$_DISCARD_2D_SEGMENTS	Segments of the graphics ($2D) content category, and any nested segments, are discarded.
DDIF$_DISCARD_T_SEGMENTS	Segments of the text ($T) content category, and any nested segments, are discarded.
DDIF$_DISCARD_TBL_SEGMENTS	Segments of the table ($TBL) content category, and any nested segments, are discarded.
DDIF$_DISCARD_PDL_SEGMENTS	Segments of the page description language ($PDL) content category, and any nested segments, are discarded.

This item list contains options only to control input processing. If you are creating a root aggregate for output processing, you must specify both an item length and an item buffer address of 0.

aggregate-type

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The type of aggregate to be created, expressed as a symbolic constant. The **aggregate-type** argument is the address of an unsigned longword that specifies the aggregate type. The only valid root aggregate type is DDIF$_DDF.

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives a value that identifies the newly created root aggregate. The **root-aggregate-handle** argument is the address of an unsigned longword that receives this root aggregate handle. This handle must be used in all subsequent operations on that root aggregate.

DESCRIPTION

The CREATE ROOT AGGREGATE routine creates a document root aggregate.

Call Format for User Allocation/Deallocation Routines

The **alloc-rtn**, **dealloc-rtn**, and **alloc-dealloc-prm** arguments are used to invoke a user routine that performs memory allocation or deallocation, and to supply an argument to that routine. The call format for one of these user routines is as follows:

user-rtn num-bytes ,base-adr ,alloc-dealloc-prm

num-bytes

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The number of bytes to allocate or free. The **num-bytes** argument is the address of an unsigned longword that contains this number of bytes. The value of **num-bytes** must be greater than zero.

base-adr

VMS usage: **address**
type: **longword (unsigned)**
access: **read only or write only**
mechanism: **by reference**

Virtual address of the first byte of memory allocated or freed. The **base-adr** argument is the address of an unsigned longword containing this base address. (This argument is write-only for a *get* routine, and read-only for a *free* routine.)

alloc-dealloc-prm

VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

User context argument. The **alloc-dealloc-prm** argument contains the value of the parameter to be passed to the user routine.

Each of these user routines must return a completion status. The VMS convention for completion codes is followed: if the low bit of the return value is clear, an error has occurred and the caller returns control to its caller; if the low bit of the return value is set, the caller continues execution.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVAGGTYP	Invalid aggregate type.
CDA$_INVITMLST	Invalid item list.

Any error returned by the memory allocation routines.

EXAMPLE

```
    .
    .
    .
aggregate_type = DDIF$_DDF;
status = cda$create_root_aggregate(0, 0, 0, 0, &aggregate_type,
                                   &root_aggregate_handle);
if (FAILURE(status)) return(status);
    .
    .
    .
```

This code segment illustrates a typical call to the CREATE ROOT AGGREGATE routine. The first four parameters are passed as zero values, indicating that the default system memory allocation and deallocation routines (LIB$GET_VM and LIB$FREE_VM) are used and that no processing options are specified. The aggregate type passed is DDIF$_DDF, which is the document root aggregate, and the root aggregate handle that is returned is used to identify that document throughout the program.

CDA$CREATE_STREAM CREATE STREAM

Opens a compound document stream for output.

FORMAT

CDA$CREATE_STREAM *alloc-rtn ,dealloc-rtn ,alloc-dealloc-prm ,put-rtn ,put-prm ,buf-len ,buf-adr ,stream-handle*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

alloc-rtn

VMS usage: **procedure**
type: **procedure entry mask**
access: **call after stack unwind**
mechanism: **by reference, procedure reference**

Address of a memory allocation routine. The **alloc-rtn** argument is the address of a procedure entry mask for this allocation routine. The calling sequence for an allocation routine is defined in the Description section of this routine. On VMS systems, if you specify 0 for this argument, LIB$GET_VM is used as the memory allocation routine.

dealloc-rtn

VMS usage: **procedure**
type: **procedure entry mask**
access: **call after stack unwind**
mechanism: **by reference, procedure reference**

Address of a memory deallocation routine. The **dealloc-rtn** argument is the address of a procedure entry mask for this deallocation routine. The calling sequence for a deallocation routine is defined in the Description section of this routine. On VMS systems, if you specify 0 for this argument, LIB$FREE_VM is used as the memory deallocation routine.

alloc-dealloc-prm

VMS usage: **context**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

User context to be passed to the memory allocation and deallocation routines. The **alloc-dealloc-prm** argument is the value of this user context. If the VMS system default memory allocation or deallocation routine is used, this value must be a zone identifier or 0 for the default zone.

put-rtn

VMS usage: **procedure**
type: **procedure entry mask**
access: **read only**
mechanism: **by reference, procedure reference**

Address of a stream *put* routine. The **put-rtn** argument is the address of a procedure entry mask for this stream *put* routine. The calling sequence for a *put* routine is defined in the Description section. If you specify 0 for this argument, the VMS RMS $PUT service is used. If you specify a value other than the default for this argument, you must also specify a value for the **put-prm** argument.

put-prm

VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

User context to be passed to the stream *put* routine. The **put-prm** argument is the value of this user context. If the VMS system default *put* routine is used, the value must be a pointer to a RAB.

buf-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Length of the buffer specified by the **buf-adr** parameter. The **buf-len** argument is the address of an unsigned longword that contains this length.

buf-adr

VMS usage: **vector_byte_unsigned**
type: **byte (unsigned)**
access: **read only**
mechanism: **by reference, array reference**

Address of a buffer that receives the output data. The **buf-adr** argument is the address of an array of unsigned bytes that make up the buffer.

stream-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the handle of the newly created stream. The **stream-handle** argument is the address of an unsigned longword that receives this stream handle. This handle must be used in all subsequent operations on that stream.

DESCRIPTION

The CREATE STREAM routine opens a compound document stream for output. The number of streams that you can open simultaneously is limited only by the amount of memory available.

Call Format for User Allocation/Deallocation Routines

The **alloc-rtn**, **dealloc-rtn**, and **alloc-dealloc-prm** arguments are used to invoke a user routine that performs memory allocation or deallocation, and to supply an argument to that routine. The call format for each of these user routines is as follows:

user-rtn num-bytes ,base-adr ,alloc-dealloc-prm

num-bytes
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**
The number of bytes to allocate or free. The **num-bytes** argument is the address of an unsigned longword that contains this number of bytes. The value of **num-bytes** must be greater than zero.

base-adr
VMS usage: **address**
type: **longword (unsigned)**
access: **read only or write only**
mechanism: **by reference**
Virtual address of the first byte of memory allocated or freed. The **base-adr** argument is the address of an unsigned longword containing this base address. (This argument is write-only for a *get* routine, and read-only for a *free* routine.)

alloc-dealloc-prm
VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**
User context argument. The **alloc-dealloc-prm** argument contains the value of the parameter to be passed to the user routine.

Call Format for User *Put* Routines

The **put-rtn** and **put-prm** arguments are used to invoke a user stream *put* routine, and to supply an argument to that routine. A *put* routine writes bytes of information to an output stream. The output buffer is initially supplied by the application through a call to the CREATE STREAM routine. Each call to the *put* routine supplies the next output buffer; therefore, the application can use any buffer management technique. The caller of the *put* routine makes no further use of the buffer described by **num-bytes** and **buf-adr**.

The call format for a user *put* routine is as follows:

put-rtn put-prm ,num-bytes ,buf-adr ,next-buf-len ,next-buf-adr

put-prm
VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**
User context argument. The **put-prm** argument is the value of the parameter to be passed to the user *put* routine.

num-bytes
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**
Number of bytes contained in the buffer. The **num-bytes** argument is the address of an unsigned longword that contains this value.

buf-adr
VMS usage: **vector_byte_unsigned**
type: **byte (unsigned)**
access: **read only**
mechanism: **by reference, array reference**
Address of the buffer. The **buf-adr** argument is the address of an array of unsigned bytes.

next-buf-len
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives the length (in bytes) of the buffer specified by **next-buf-adr**. The **next-buf-len** argument is the address of an unsigned longword that receives this length.

next-buf-adr
VMS usage: **address**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives the address of a buffer that will receive further output data. The **next-buf-adr** argument is the address of an unsigned longword that receives this address. **Next-buf-adr** may simply be the current buffer, or a different buffer.

Each of these user routines must return a completion status. The VMS convention for completion codes is followed: if the low bit of the return value is clear, an error has occurred and the caller returns control to its calling procedure; if the low bit of the return value is set, the caller continues execution.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.

Any error returned by the memory allocation routines.

CDA$CREATE_TEXT_FILE CREATE TEXT FILE

Creates a standard text file for output.

FORMAT

CDA$CREATE_TEXT_FILE *file-spec-len ,file-spec ,default-file-spec-len ,default-file-spec ,result-file-spec-len ,result-file-spec ,result-file-ret-len ,text-file-handle*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

file-spec-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Length (in bytes) of the string specified by the **file-spec** argument. The **file-spec-len** argument is the address of an unsigned longword that contains this length.

file-spec

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by reference**

File specification of the text file to be created for output. The **file-spec** argument is the address of a character string containing this file specification.

default-file-spec-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Length (in bytes) of the string specified by **default-file-spec**. The **default-file-spec-len** argument is the address of an unsigned longword that contains this default file specification length. If you specify 0 for this parameter, no default file specification is used.

default-file-spec

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by reference**

Default file specification. The **default-file-spec** argument is the address of a character string that contains this default file specification. If you specify 0 for this parameter, no default file specification is used. The string should consist only of a file type in lowercase characters.

result-file-spec-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Length (in bytes) of the buffer specified by **result-file-spec**. The **result-file-spec-len** argument is the address of an unsigned longword that contains this buffer length. If you specify 0 for this parameter, the length of the resultant file specification is not returned.

result-file-spec

VMS usage: **char_string**
type: **character string**
access: **write only**
mechanism: **by reference**

Receives the resultant file specification. The **result-file-spec** argument is the address of a character string that receives this file specification. This file specification is the result of a VMS RMS $CREATE operation.

result-file-ret-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the actual length (in bytes) of the resultant file specification. The **result-file-ret-len** argument is the address of an unsigned longword that receives the actual length of the resultant file specification. If you specify 0 for this parameter, the actual length of the resultant file specification is not returned.

text-file-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the handle of the text file. The **text-file-handle** argument is the address of an unsigned longword that receives this text file handle. This handle must be used in all subsequent operations on that text file.

DESCRIPTION

The CREATE TEXT FILE routine creates a standard text file for output.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.

Any error returned by the memory allocation routines.

Any error returned by the file routines.

CDA$DELETE_AGGREGATE DELETE AGGREGATE

Destroys an aggregate and all of its substructure. If the specified aggregate is part of a sequence, the aggregate is cut from the sequence before being destroyed.

FORMAT

CDA$DELETE_AGGREGATE *root-aggregate-handle ,aggregate-handle*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate associated with the aggregate to be deleted. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

You must use identical memory management procedures when creating and deleting an aggregate, to ensure consistent treatment of memory allocation and deallocation.

aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the aggregate to be destroyed. The **aggregate-handle** argument is the address of an unsigned longword that contains this aggregate identifier.

DESCRIPTION

The DELETE AGGREGATE routine destroys an aggregate and all of its substructure. If the specified aggregate is part of a sequence, the aggregate is cut from the sequence before being destroyed. Note that the specified aggregate handle, and the handles of any subaggregates linked to the specified aggregate either directly or indirectly, are invalid after a call to this routine.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVAGGTYP	Invalid aggregate type.

Any error returned by the memory deallocation routines.

CDA$DELETE_ROOT_AGGREGATE DELETE ROOT AGGREGATE

Destroys a document root aggregate and all of its substructure.

FORMAT

CDA$DELETE_ROOT_AGGREGATE
root-aggregate-handle

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate to be deleted. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

DESCRIPTION

The DELETE ROOT AGGREGATE routine destroys a document root aggregate and all of its associated substructure. The root aggregate and its substructure form a tree structure, so that when the root aggregate is deleted, any aggregates attached to that root aggregate are also deleted. The **root-aggregate-handle**, as well as the handles of any aggregates that are linked to the root aggregate either directly or indirectly, are invalid after a call to this routine.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.

Any error returned by the memory deallocation routines.

CDA$ENTER_SCOPE ENTER SCOPE

Opens a document scope for incremental writing.

FORMAT

CDA$ENTER_SCOPE *root-aggregate-handle ,stream-handle ,scope-code [,aggregate-handle]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate associated with the document content to be incrementally written. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

stream-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the stream associated with the document to be written. The **stream-handle** argument is the address of an unsigned longword that contains this stream handle. This handle is returned by a call to either the CREATE FILE routine or the CREATE STREAM routine.

scope-code

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Symbolic constant identifying the scope to be opened. The **scope-code** argument is the address of an unsigned longword containing this code. Valid values are as follows:

Code	Meaning
DDIF$K_DOCUMENT_SCOPE	Document scope
DDIF$K_CONTENT_SCOPE	Content scope
DDIF$K_SEGMENT_SCOPE	Segment scope

aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of an aggregate of the appropriate type, if required by the scope code specified. The **aggregate-handle** argument is the address of an unsigned longword that contains this aggregate handle.

The aggregate must be completely populated, except that its content sequence must be empty. The DDIF scoped sections that require that the **aggregate-handle** be specified are as follows:

Scope	Value of Aggregate-Handle
DDIF$K_SEGMENT_SCOPE	**Aggregate-handle** is the handle of an aggregate of type DDIF$_SEG.

DESCRIPTION

The ENTER SCOPE routine lets you open a particular document scope for incremental writing. The types of scopes that you can open for a document are the following:

- Document scope
- Content scope
- Segment scope

When performing incremental writing, you should perform the following steps:

1. Call the ENTER SCOPE routine, specifying **scope-code** as DDIF$K_DOCUMENT_SCOPE.
2. Write an aggregate of type DDIF$_DSC.
3. Write an aggregate of type DDIF$_DHD.
4. Call the ENTER SCOPE routine, specifying **scope-code** as DDIF$K_CONTENT_SCOPE.
5. Write a root segment of type DDIF$_SEG. The root segment is a top level segment that contains the document content. This document content can consist of content aggregates as well as nested segments. If the document contains only one segment, that segment is the root segment and it contains all of the document content. If the document contains multiple segments, they must be nested within a root segment.

You can use either of the methods outlined below to create the root segment. Because the first method requires that the entire segment be completed before calling the PUT AGGREGATE routine, once you select that method you must continue to use that method while writing all of the document content. If you select the second method, you can use either method to write any nested segments. Again, if while writing nested segments, you select the first method, you must continue to use that method, and so on.

a. Call the PUT AGGREGATE routine with a completed aggregate of type DDIF$_SEG, whose DDIF$_SEG_CONTENT item references a sequence of aggregates that make up the entire content for that segment, including any nested segments. Using this method, you need only call the PUT AGGREGATE routine once, because the DDIF$_SEG aggregate written in the call to PUT AGGREGATE is already completely populated.

b. Call the ENTER SCOPE routine, specifying **scope-code** as DDIF$K_SEGMENT_SCOPE, with a completed aggregate of type DDIF$_SEG whose DDIF$_SEG_CONTENT item is empty. You can then call the PUT AGGREGATE routine for each aggregate that makes up the segment content, in order. Once that segment and all its nested segments have been output, call the LEAVE SCOPE routine, specifying **scope-code** as DDIF$K_SEGMENT_SCOPE to complete that segment.

6 Call the LEAVE SCOPE routine, specifying **scope-code** as DDIF$K_CONTENT_SCOPE.

7 Call the LEAVE SCOPE routine, specifying **scope-code** as DDIF$K_DOCUMENT_SCOPE.

When you call the ENTER SCOPE routine with **scope-code** specified as DDIF$K_SEGMENT_SCOPE, you can write aggregates of the following types within the segment, provided that the appropriate restrictions on content types within content categories are observed:

Aggregate Type	Meaning
DDIF$_SEG	Document segment
DDIF$_TXT	Text content
DDIF$_HRD	Hard directive
DDIF$_SFT	Soft directive
DDIF$_LIN	Polyline content
DDIF$_ARC	Arc content
DDIF$_BEZ	Bézier curve content
DDIF$_IMG	Image content
DDIF$_CRF	Content reference
DDIF$_EXT	External content
DDIF$_PVT	Private content

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVSCOCOD	Invalid scope code.

Any errors returned by the file routines.

EXAMPLES

1

```
        .
        .
        .
/* Get the document from the front end using the aggregate method */
while (SUCCESS(status = cda$convert_aggregate (&root_aggregate_handle,
                                               fre_handle,
                                               &aggregate_handle,
                                               &aggregate_type)))
{
        switch (aggregate_type)
        {
            /* If the aggregate type is DDIF$_DSC, the document
               descriptor aggregate, then enter document scope
               and write the aggregate to the stream */
            case DDIF$_DSC:
                scope = DDIF$K_DOCUMENT_SCOPE;
                status = cda$enter_scope (&root_aggregate_handle,
                                          &stream_handle,
                                          &scope);
                if (!SUCCESS(status))
                        CLEANUP (status);

                status = cda$put_aggregate (&root_aggregate_handle,
                                            &stream_handle,
                                            &aggregate_handle);
                if (!SUCCESS(status))
                        CLEANUP (status);
                break;

            /* If the aggregate type is DDIF$_DHD, the document
               header aggregate, then simply write the aggregate
               to the stream, since we're already in the document
               scope */
            case DDIF$_DHD:
                status = cda$put_aggregate (&root_aggregate_handle,
                                            &stream_handle,
                                            &aggregate_handle);
                if (!SUCCESS(status))
                        CLEANUP (status);

                scope = DDIF$K_CONTENT_SCOPE;
                status = cda$enter_scope (&root_aggregate_handle,
                                          &stream_handle,
                                          &scope);
                if (!SUCCESS(status))
                        CLEANUP (status);
                break;
```

```
            /* If the aggregate type is DDIF$_SEG, the segment
               aggregate, then enter the segment scope and write
               the aggregate to the stream */
            case DDIF$_SEG:
                scope = DDIF$K_SEGMENT_SCOPE;
                status = cda$enter_scope (&root_aggregate_handle,
                                          &stream_handle,
                                          &scope,
                                          &aggregate_handle);
                if (!SUCCESS(status))
                        CLEANUP (status);
                break;

            /* If the aggregate type is DDIF$_EOS, end of
               segment aggregate, then leave the segment scope */
            case DDIF$_EOS:
                scope = DDIF$K_SEGMENT_SCOPE;
                status = cda$leave_scope (&root_aggregate_handle,
                                          &stream_handle,
                                          &scope);
                if (!SUCCESS(status))
                        CLEANUP (status);
                break;

            /* For any other aggregate type, simply write the
               aggregate to the stream */
            default:
                status = cda$put_aggregate (&root_aggregate_handle,
                                            &stream_handle,
                                            &aggregate_handle);
                if (!SUCCESS(status))
                        CLEANUP (status);
                break;

        }

        /* Delete the aggregate(s) just processed */
        status = cda$delete_aggregate (&root_aggregate_handle,
                                       &aggregate_handle);
        if (!SUCCESS(status))
                CLEANUP (status);
}

/* Once all aggregates are processed, leave the content scope and
   the document scope */

   .
   .
   .
```

This example illustrates the use of the ENTER SCOPE and LEAVE SCOPE routines to read an input document using the aggregate (incremental) method.

The following example also illustrates the incremental method of creating a document, using both of the methods outlined for writing nested segments.

```
2   /*
       This is an example of using the incremental method to create a document
       with nested segments being output using different options.
    */

    #include <cda$def.h>
    #include <ddif$def.h>
    #define FAILURE(x) (((x) & 1) == 0)

    main()
    {
    unsigned long   status;
    unsigned long   aggregate_type;
    unsigned long   aggregate_handle;
    unsigned long   prev_aggregate_handle;
    unsigned long   aggregate_item;
    unsigned long   aggregate_index;
    unsigned long   add_info;
    unsigned long   spec_length;
    unsigned long   result_length;
    unsigned char   result_buffer[255];
    unsigned long   stream_handle;
    unsigned long   file_handle;
    unsigned long   root_aggregate_handle;
    unsigned long   segment_handle;
    unsigned long   integer_value;
    unsigned char   byte_value;
    unsigned long   buffer_length;
    unsigned long   scope_code;

            /* Create the root aggregate */
            aggregate_type = DDIF$_DDF;
            status = cda$create_root_aggregate(0, 0, 0, 0, &aggregate_type,
                     &root_aggregate_handle);
            if (FAILURE(status)) return(status);

            /* Create the file */
            spec_length = 9;
            result_length = sizeof(result_buffer);
            status = cda$create_file(&spec_length, "test.ddif", 0, 0,
                                     0, 0, 0,
                                     &root_aggregate_handle, &result_length,
                                     &result_buffer[0], &result_length,
                                     &stream_handle, &file_handle);
            if (FAILURE(status)) return(status);

            /* Enter Document Scope */
            scope_code = DDIF$K_DOCUMENT_SCOPE;
            status = cda$enter_scope(&root_aggregate_handle, &stream_handle,
                                     &scope_code);
            if (FAILURE(status)) return(status);

            /* Create, populate, put, and delete the descriptor aggregate */
            aggregate_type = DDIF$_DSC;
            status = cda$create_aggregate(&root_aggregate_handle,
                                          &aggregate_type, &aggregate_handle);
            if (FAILURE(status)) return(status);

            aggregate_item = DDIF$_DSC_MAJOR_VERSION;
            buffer_length = sizeof(integer_value);
            integer_value = 1;
            status = cda$store_item(&root_aggregate_handle, &aggregate_handle,
                           &aggregate_item, &buffer_length, &integer_value);
            if (FAILURE(status)) return(status);
```

```
aggregate_item = DDIF$_DSC_MINOR_VERSION;
buffer_length = sizeof(integer_value);
integer_value = 0;
status = cda$store_item(&root_aggregate_handle, &aggregate_handle,
                        &aggregate_item, &buffer_length, &integer_value);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_DSC_PRODUCT_IDENTIFIER;
buffer_length = 4;
status = cda$store_item(&root_aggregate_handle, &aggregate_handle,
                           &aggregate_item, &buffer_length, "Test");
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_DSC_PRODUCT_NAME;
buffer_length = 19;
add_info = CDA$K_ISO_LATIN1;
aggregate_index = 0;
status = cda$store_item(&root_aggregate_handle, &aggregate_handle,
                  &aggregate_item, &buffer_length,
                  "Example Application", &aggregate_index, &add_info);
if (FAILURE(status)) return(status);

status = cda$put_aggregate(&root_aggregate_handle,
                              &stream_handle, &aggregate_handle);
if (FAILURE(status)) return(status);

status = cda$delete_aggregate(&root_aggregate_handle,
                                 &aggregate_handle);
if (FAILURE(status)) return(status);

/* Create, populate, put, and delete the header aggregate. */
aggregate_type = DDIF$_DHD;
status = cda$create_aggregate(&root_aggregate_handle,
                                 &aggregate_type, &aggregate_handle);
if (FAILURE(status)) return(status);
prev_aggregate_handle = aggregate_handle;

/* Store header items here */

status = cda$put_aggregate(&root_aggregate_handle, &stream_handle,
                              &aggregate_handle);
if (FAILURE(status)) return(status);

status = cda$delete_aggregate(&root_aggregate_handle,
                                 &aggregate_handle);
if (FAILURE(status)) return(status);

/* Enter Content Scope */
scope_code = DDIF$K_CONTENT_SCOPE;
status = cda$enter_scope(&root_aggregate_handle, &stream_handle,
                            &scope_code);
if (FAILURE(status)) return(status);

/* Create the "root sement" aggregate, and fill it in except for
   the content.  This will be output using cda$enter_scope, and
   its contents will be output incrementally.
 */
aggregate_type = DDIF$_SEG;
status = cda$create_aggregate(&root_aggregate_handle,
                     &aggregate_type, &aggregate_handle);
if (FAILURE(status)) return(status);
segment_handle = aggregate_handle;
```

```
/* Fill in any items needed at the top level. */
aggregate_type = DDIF$_SGA;
status = cda$create_aggregate(&root_aggregate_handle,
                              &aggregate_type, &aggregate_handle);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_SEG_SPECIFIC_ATTRIBUTES;
buffer_length = sizeof(aggregate_handle);
status = cda$store_item(&root_aggregate_handle, &segment_handle,
                  &aggregate_item, &buffer_length, &aggregate_handle);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_SGA_CONTENT_CATEGORY;
add_info = DDIF$K_T_CATEGORY;
status = cda$store_item(&root_aggregate_handle, &aggregate_handle,
                        &aggregate_item, 0, 0, 0, &add_info);
if (FAILURE(status)) return(status);

/* Enter Segment Scope.  This requires the segment aggregate handle,
   and causes the segment aggregate to be output.  */
scope_code = DDIF$K_SEGMENT_SCOPE;
status = cda$enter_scope(&root_aggregate_handle, &stream_handle,
                         &scope_code, &segment_handle);
if (FAILURE(status)) return(status);

/* Delete the segment aggregate */
status = cda$delete_aggregate(&root_aggregate_handle, &segment_handle);
if (FAILURE(status)) return(status);

/* Incrementally, create the content aggregates and put them out. */
aggregate_type = DDIF$_TXT;
status = cda$create_aggregate(&root_aggregate_handle, &aggregate_type,
                              &aggregate_handle);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_TXT_CONTENT;
buffer_length = 5;
status = cda$store_item(&root_aggregate_handle, &aggregate_handle,
                        &aggregate_item, &buffer_length, "Hello");
if (FAILURE(status)) return(status);

status = cda$put_aggregate(&root_aggregate_handle, &stream_handle,
                           &aggregate_handle);
if (FAILURE(status)) return(status);

/* Delete the text aggregate */
status = cda$delete_aggregate(&root_aggregate_handle,
                              &aggregate_handle);
if (FAILURE(status)) return(status);

/* The next content element is a segment
 * Create a sement aggregate, link all it's content to it, and output
 * the aggregate.  (This segment does not use cda$enter_scope.)
 */
aggregate_type = DDIF$_SEG;
status = cda$create_aggregate(&root_aggregate_handle,
                        &aggregate_type, &aggregate_handle);
if (FAILURE(status)) return(status);
segment_handle = aggregate_handle;

aggregate_type = DDIF$_SGA;
status = cda$create_aggregate(&root_aggregate_handle,
                      &aggregate_type, &aggregate_handle);
if (FAILURE(status)) return(status);
```

```
aggregate_item = DDIF$_SEG_SPECIFIC_ATTRIBUTES;
buffer_length = sizeof(aggregate_handle);
status = cda$store_item(&root_aggregate_handle, &segment_handle,
                        &aggregate_item, &buffer_length,
                        &aggregate_handle);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_SGA_CONTENT_CATEGORY;
add_info = DDIF$K_T_CATEGORY;
status = cda$store_item(&root_aggregate_handle, &aggregate_handle,
                        &aggregate_item, 0, 0, 0, &add_info);
if (FAILURE(status)) return(status);

/* Create content aggregates, and link them to
 * the segment aggregate.
 */
aggregate_type = DDIF$_TXT;
status = cda$create_aggregate(&root_aggregate_handle, &aggregate_type,
                              &aggregate_handle);
if (FAILURE(status)) return(status);
prev_aggregate_handle = aggregate_handle;

aggregate_item = DDIF$_SEG_CONTENT;
buffer_length = sizeof(aggregate_handle);
status = cda$store_item(&root_aggregate_handle, &segment_handle,
                        &aggregate_item, &buffer_length,
                        &aggregate_handle);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_TXT_CONTENT;
buffer_length = 5;
status = cda$store_item(&root_aggregate_handle, &aggregate_handle,
                        &aggregate_item, &buffer_length,
                        "There");
if (FAILURE(status)) return(status);

aggregate_type = DDIF$_HRD;
status = cda$create_aggregate(&root_aggregate_handle, &aggregate_type,
                              &aggregate_handle);
if (FAILURE(status)) return(status);

cda$insert_aggregate(&aggregate_handle, &prev_aggregate_handle);

aggregate_item = DDIF$_HRD_DIRECTIVE;
buffer_length = sizeof(integer_value);
integer_value = DDIF$K_DIR_NEW_PAGE;
status = cda$store_item(&root_aggregate_handle, &aggregate_handle,
                        &aggregate_item, &buffer_length,
                        &integer_value);
if (FAILURE(status)) return(status);

/* Output the segment aggregate (Since the content is attached,
 * it is output also.)
 */
status = cda$put_aggregate(&root_aggregate_handle, &stream_handle,
                           &segment_handle);
if (FAILURE(status)) return(status);


/* Delete the segment aggregate and all aggregates
 * attached to it.
 */
status = cda$delete_aggregate(&root_aggregate_handle, &segment_handle);
if (FAILURE(status)) return(status);

/* Output more content aggregates within the root segment */
```

```
        /* Leave Segment Scope.  This is for the segment that was output
           using cda$enter_scope.  */
        scope_code = DDIF$K_SEGMENT_SCOPE;
        status = cda$leave_scope(&root_aggregate_handle, &stream_handle,
                                 &scope_code);
        if (FAILURE(status)) return(status);

        /* Leave Content Scope */
        scope_code = DDIF$K_CONTENT_SCOPE;
        status = cda$leave_scope(&root_aggregate_handle, &stream_handle,
                                 &scope_code);
        if (FAILURE(status)) return(status);

        /* Leave Document Scope */
        scope_code = DDIF$K_DOCUMENT_SCOPE;
        status = cda$leave_scope(&root_aggregate_handle, &stream_handle,
                                 &scope_code);
        if (FAILURE(status)) return(status);

        /* Close the file */
        status = cda$close_file(&stream_handle, &file_handle);
        if (FAILURE(status)) return(status);

        /* Delete the root aggregate */
        status = cda$delete_root_aggregate(&root_aggregate_handle);
        if (FAILURE(status)) return(status);

        return 1;
}
```

This example illustrates the use of both methods of incremental writing: using the PUT AGGREGATE routine with a completed segment or using ENTER SCOPE and incrementally writing the segment's content. This program creates a DDIF file whose analysis would appear as follows:

```
3   DDIF_DOCUMENT
    {
     DDF_DESCRIPTOR
     {
      DSC_MAJOR_VERSION  1  ! Longword Integer
      DSC_MINOR_VERSION  0  ! Longword Integer
      DSC_PRODUCT_IDENTIFIER  "%H54657374"  ! Byte string = "Test"
      DSC_PRODUCT_NAME
      (
       ISO_LATIN1  "Example Application"
      )
     }
     DDF_HEADER
     {
     }
     DDF_CONTENT
     {
      SEG_SPECIFIC_ATTRIBUTES
      {
       SGA_CONTENT_CATEGORY  T_CATEGORY  "$T"
      }
      SEG_CONTENT
      {
       TXT_CONTENT  "%H48656C6C6F"  ! Byte string = "Hello"
      }
      {
       SEG_SPECIFIC_ATTRIBUTES
       {
        SGA_CONTENT_CATEGORY  T_CATEGORY  "$T"
       }
```

```
    SEG_CONTENT
    {
     TXT_CONTENT  "%H5468657265"  ! Byte string = "There"
    }
    {
     HRD_DIRECTIVE  DIR_NEW_PAGE  ! Integer = 1
    }
   }
  }
}
```

CDA$ERASE_ITEM ERASE ITEM

Erases (sets to empty) the contents of an item within an aggregate. If you erase an item that is indexed, the index of each subsequent item (each item with a higher index) decreases by 1.

FORMAT

CDA$ERASE_ITEM *root-aggregate-handle*
,aggregate-handle
,aggregate-item
[,aggregate-index]

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate of which the aggregate containing the item is a part. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

You must use identical memory management procedures when creating and deleting an aggregate, to ensure consistent treatment of memory allocation and deallocation.

aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the aggregate containing the item to be erased. The **aggregate-handle** argument is the address of an unsigned longword that contains this aggregate handle.

aggregate-item

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifying code of the item to be erased, expressed as a symbolic constant. The **aggregate-item** argument is the address of an unsigned longword that contains this code. The DDIF aggregate item symbolic constants are

defined in the file DDIF$DEF.SDL and are discussed in Chapter 6 and Appendix D.

aggregate-index

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Index of the item to be erased (relative to 0). The optional **aggregate-index** argument is the address of an unsigned longword that contains this index. This argument is required whenever the notation "Array of" appears in the data type of the specified item handle. Otherwise, this argument is ignored and may be omitted. If an address of 0 is specified, all of the array elements in the item are erased.

DESCRIPTION

The ERASE ITEM routine erases (sets to empty) the contents of an item within an aggregate. If you erase an item that is indexed, the index of each subsequent item (each item with a higher index) decreases by 1. If you specify 0, all array elements in the item are erased.

Note that if you erase an item that contains the handle of a subaggregate, the subaggregate is erased in addition to the item.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVAGGTYP	Invalid aggregate type.
CDA$_INVITMCOD	Invalid item code.
CDA$_EMPTY	Item is empty.
CDA$_INDEX	Index exceeds array bounds.
CDA$_VAREMPTY	Variant item is empty.
CDA$_VARINDEX	Variant index exceeds bounds.
CDA$_VARVALUE	Variant value is undefined.

CDA$FIND_DEFINITION FIND DEFINITION

Looks up the specified definition in a list of definitions.

FORMAT

CDA$FIND_DEFINITION *root-aggregate-handle ,aggregate-type ,buf-len ,buf-adr ,aggregate-handle*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate with which the definition aggregate being searched for is associated. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

aggregate-type

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The type of definition aggregate being searched for, expressed as a symbolic constant. The **aggregate-type** argument is the address of an unsigned longword that contains this symbolic constant. The DDIF aggregate type symbolic constants are defined in the file DDIF$DEF.SDL and are discussed in Chapter 6 and Appendix D.

buf-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Length of the buffer (in bytes) specified by **buf-adr**. The **buf-len** argument is the address of an unsigned longword that contains this buffer length. This buffer is used as a selector value to select the desired definition from the list of definitions. For aggregate types DDIF$_FTD, DDIF$_LSD, DDIF$_PHD, DDIF$_ERF, and DDIF$_PTD, the buffer length must be 4. For a DDIF$_ERF aggregate, the buffer contains a longword value that indexes the external reference (beginning at 1).

buf-adr

VMS usage: **vector_byte_unsigned**
type: **byte (unsigned)**
access: **read only**
mechanism: **by reference, array reference**

The buffer that contains the selector value used to indicate the desired definition from the list of definitions. The **buf-adr** argument is the address of an array of unsigned bytes that comprise this buffer. The definition aggregate types DDIF$_FTD, DDIF$_LSD, DDIF$_PHD, DDIF$_ERF, and DDIF$_PTD are identified in a series of definitions by a unique number. Therefore, for these aggregate types, the **buf-adr** value must be a longword. For aggregate types DDIF$_CTD, DDIF$_TYD, and DDIF$_SGB, which are assigned string labels, the value must be a string.

aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives a value that identifies the newly located definition aggregate. The **aggregate-handle** argument is the address of an unsigned longword that receives this aggregate handle. This handle must be used in all subsequent operations on that aggregate.

DESCRIPTION

The FIND DEFINITION routine looks up the specified definition in a series of definition aggregates. For example, if you have several font definition (DDIF$_FTD) aggregates and you want to retrieve the definition of the font identified by the index 3, you would invoke this routine, specifying the **aggregate-type** as DDIF$_FTD and the selector value (**buf-adr**) as 3. The aggregate types that can be specified for this routine are as follows:

DDIF$_CTD	Content definition aggregate
DDIF$_ERF	External reference aggregate
DDIF$_FTD	Font definition aggregate
DDIF$_LSD	Line style definition aggregate
DDIF$_PHD	Path definition aggregate
DDIF$_PTD	Pattern definition aggregate
DDIF$_SGB	Segment bindings aggregate
DDIF$_TYD	Type definition aggregate

In order for this routine to return the correct information, you must have specified one or more of the following processing options in the call to the CREATE ROOT AGGREGATE routine:

- DDIF$_INHERIT_ATTRIBUTES
- DDIF$_EVALUATE_CONTENT
- DDIF$_RETAIN_DEFINITIONS

This routine is only valid when you are using the aggregate (incremental) method of document conversion, because the definition being determined is dependent upon the current location in the document. If you call this routine when you are performing document method conversion, the current position is the top of the document, so that no definition is available.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVAGGTYP	Invalid aggregate type.
CDA$_INVBUFLEN	Invalid buffer length.
CDA$_DEFNOTFOU	Definition not found.

CDA$FIND_TRANSFORMATION FIND TRANSFORMATION

Returns the current transformation matrix values.

FORMAT

CDA$FIND_TRANSFORMATION
root-aggregate-handle ,transformation

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

transformation

VMS usage: **address**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the address of a vector of nine F_floating elements. The **transformation** argument is the address of an unsigned longword that receives the address of this array.

The elements of this vector specify the current content transformation in column order. For example, the elements of the following array would be returned in the order (A,B,C,D,E,F,G,H,I).

```
A   D   G
B   E   H
C   F   I
```

DESCRIPTION

The FIND TRANSFORMATION routine returns the current values of the transformation matrix. In order for this routine to return the correct information, you must have specified one or more of the following processing options in the call to the CREATE ROOT AGGREGATE routine:

- DDIF$_INHERIT_ATTRIBUTES
- DDIF$_EVALUATE_CONTENT
- DDIF$_RETAIN_DEFINITIONS

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_DEFNOTFOU	Definition not found.

CDA$FLUSH_STREAM FLUSH STREAM

Flushes the contents of the stream and ensures that the data has been physically transferred to the receiving medium.

FORMAT

CDA$FLUSH_STREAM *stream-handle ,flush-rtn ,flush-prm*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

stream-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the output stream to be flushed. The **stream-handle** argument is the address of an unsigned longword that contains this stream handle. This handle is returned by a call to the CREATE STREAM routine.

flush-rtn

VMS usage: **procedure**
type: **procedure entry mask**
access: **call after stack unwind**
mechanism: **by reference, procedure reference**

Address of a stream *flush* routine. The **flush-rtn** argument is the address of a procedure entry mask for this stream *flush* routine. The calling sequence for a *flush* routine is defined in the Description section of this routine. If you specify 0 for this argument, the VMS RMS $FLUSH service is used. If you specify a value other than the default for this argument, you must also specify a value for the **flush-prm** argument.

flush-prm

VMS usage: **context**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

User context to be passed to the stream *flush* routine. The **flush-prm** argument is the value of this user context. This argument should contain the value of the **put-prm** argument passed in a call to the CREATE STREAM routine. If the VMS system default flush procedure is used, the value must be a pointer to a RAB.

DESCRIPTION

The FLUSH STREAM routine writes any buffered data to an output stream and ensures that the data has been physically transferred to the receiving medium.

Call Format for User *Flush* Routines

The **flush-rtn** and **flush-prm** arguments are used to invoke a user stream *flush* routine, and to supply an argument to that routine. The call format for this user routine is as follows:

flush-rtn flush-prm

flush-prm
VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**
User context argument. The **flush-prm** argument is the value of the parameter to be passed to the user flush routine.

This routine must return a completion status. The VMS convention for completion codes is followed: if the low bit of the return value is clear, an error has occurred and the caller returns control to its caller; if the low bit of the return value is set, the caller continues execution.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.

Any error returned by the file routines.

CDA$GET_AGGREGATE GET AGGREGATE

Reads the next aggregate from the specified stream.

FORMAT

CDA$GET_AGGREGATE *root-aggregate-handle*
,stream-handle
,aggregate-handle
,aggregate-type

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate associated with the aggregate to be read. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

When reading aggregates using this routine, you must use the same value for **root-aggregate-handle** consistently to read all the aggregates in the compound document. Once you have read all of the aggregates, you cannot specify the same **root-aggregate-handle** again when calling this routine.

stream-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the stream from which the aggregate is to be read. The **stream-handle** argument is the address of an unsigned longword that contains this stream handle. This handle is returned by a call to either the OPEN FILE routine or the OPEN STREAM routine.

aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the handle of the retrieved aggregate. The **aggregate-handle** argument is the address of an unsigned longword that receives this

aggregate handle. This aggregate handle is used to identify the retrieved aggregate to any other aggregate transfer procedure.

aggregate-type

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the aggregate type. The **aggregate-type** argument is the address of an unsigned longword that receives this aggregate type. The DDIF aggregate type symbolic codes are defined in the file DDIF$DEF.SDL and are described in Chapter 6 and Appendix D.

Valid aggregate types are as follows:

Aggregate Type	Meaning
DDIF$_DSC	Document descriptor
DDIF$_DHD	Document header
DDIF$_SEG	Document segment
DDIF$_TXT	Text content
DDIF$_GTX	General text content
DDIF$_HRD	Hard directive
DDIF$_SFT	Soft directive
DDIF$_HRV	Hard value directive
DDIF$_SFV	Soft value directive
DDIF$_BEZ	Bézier curve content
DDIF$_LIN	Polyline content
DDIF$_ARC	Arc content
DDIF$_FAS	Fill area set content
DDIF$_IMG	Image content
DDIF$_CRF	Content reference
DDIF$_EXT	External content
DDIF$_PVT	Private content
DDIF$_GLY	Layout galley
DDIF$_EOS	End of segment

If the aggregate type is DDIF$_EOS (end of segment), the **aggregate-handle** is 0.

DESCRIPTION

The GET AGGREGATE routine reads the next aggregate from a specified stream. This routine is used by a front end to read the next aggregate from a compound document file. Note that the aggregate returned is not part of a sequence.

The GET AGGREGATE routine reads the aggregates in a document in a hierarchical fashion. That is, whenever GET AGGREGATE encounters a segment, it descends to the next level of hierarchy and reads the contents of that segment before reading the remaining content of the parent segment. The GET AGGREGATE routine only returns to the parent segment's level of hierarchy when it encounters a DDIF$_EOS (end of segment) aggregate to indicate that the nested segment is completed. These rules can be generalized as follows:

- If the aggregate being read is a content aggregate, the aggregate is simply returned and the next aggregate returned is the next aggregate in the segment.
- If the aggregate being read is a segment aggregate (DDIF$_SEG), the content nested in the segment is returned, using these same ordering rules, followed by a dummy DDIF$_EOS (end of segment) aggregate to indicate the end of the nested segment. Once the nested segment and its content have been returned and the end of the segment has been indicated, the next aggregate read is the next aggregate in the (current) segment.

Note: All segments must be completed by a DDIF$_EOS aggregate.

Following these generalized rules, if a document contains a document root aggregate (DDIF$_DDF), a document descriptor (DDIF$_DSC), a document header (DDIF$_DHD), and a root segment (DDIF$_SEG) with text content (DDIF$_TXT), a nested segment (DDIF$_SEG), and Bézier content (DDIF$_BEZ), where the segment nested under the root segment contains arc content (DDIF$_ARC), the aggregates returned by consecutive calls to GET AGGREGATE would be as follows:

1. DDIF$_DSC
2. DDIF$_DHD
3. DDIF$_SEG (root segment)
4. DDIF$_TXT
5. DDIF$_SEG (segment with nested arc content)
6. DDIF$_ARC (nested arc content aggregate)
7. DDIF$_EOS (dummy aggregate indicating end of segment with nested arc content)
8. DDIF$_BEZ (Bézier content)
9. DDIF$_EOS (dummy aggregate indicating end of root segment)

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_ENDOFDOC	End of document.
CDA$_INVDOC	Invalid document content.

Any error returned by the memory allocation routines.

Any error returned by the file routines.

CDA$GET_ARRAY_SIZE GET ARRAY SIZE

Determines the number of elements present in an array-valued aggregate item.

FORMAT

CDA$GET_ARRAY_SIZE *aggregate-handle ,aggregate-item ,array-size*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the aggregate containing the array-valued item. The **aggregate-handle** argument is the address of an unsigned longword containing this aggregate identifier.

aggregate-item

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifying code of the array-valued aggregate item, expressed as a symbolic constant. The **aggregate-item** argument is the address of an unsigned longword that contains this code. The DDIF aggregate item symbolic constants are defined in the module DDIF$DEF.SDL and are defined in Chapter 6 and Appendix D.

array-size

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the number of elements present in the array-valued item. The **array-size** argument is the address of an unsigned longword that receives this array size. Because the index is zero-based, this number is equal to one more than the value of the highest valid aggregate index.

DESCRIPTION

The GET ARRAY SIZE routine determines the number of elements present in an array-valued aggregate item.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVAGGTYP	Invalid aggregate type.
CDA$_INVITMCOD	Invalid item code.
CDA$_EMPTY	Item is empty.

CDA$GET_DOCUMENT GET DOCUMENT

Reads an entire compound document from the specified stream.

FORMAT

CDA$GET_DOCUMENT *root-aggregate-handle ,stream-handle*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate associated with the document to be read. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

Once you read an entire document, you cannot call the GET DOCUMENT routine specifying the same root aggregate handle again. That is, you can only read a document associated with a particular root aggregate once.

stream-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the stream from which the document is to be read. The **stream-handle** argument is the address of an unsigned longword that contains this stream handle. This handle is returned by a call to either the OPEN FILE routine or the OPEN STREAM routine.

DESCRIPTION

The GET DOCUMENT routine reads an entire document from the specified stream. This routine is used by a front end module to read an entire compound document file into memory.

Upon completion of the call to this routine, the entire document is present in aggregates that are linked from the document root aggregate.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVAGGTYP	Invalid aggregate type.
CDA$_INVDOC	Invalid document content.

Any error returned by the memory allocation routines.

Any error returned by the file routines.

CDA$GET_EXTERNAL_ENCODING
GET EXTERNAL ENCODING

Reads the value of an external encoding from the specified stream and stores it as the value of the DDIF$_EXT_ENCODING item in a DDIF$_EXT aggregate.

FORMAT

CDA$GET_EXTERNAL_ENCODING
root-aggregate-handle ,stream-handle ,aggregate-handle

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

stream-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the stream containing the external encoding. The **stream-handle** argument is the address of an unsigned longword that contains this stream handle. This handle is returned by a call to either the OPEN FILE routine or the OPEN STREAM routine.

aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Identifier of an aggregate of type DDIF$_EXT. The **aggregate-handle** argument is the address of an unsigned longword that contains this aggregate handle. The external encoding value that is read from the stream is written to the DDIF$_EXT_ENCODING item in this aggregate, and the DDIF$_EXT aggregate becomes the root aggregate for the external document.

DESCRIPTION

The GET EXTERNAL ENCODING routine reads the value of an external encoding and stores the value in the DDIF$_EXT_ENCODING item of the DDIF$_EXT aggregate specified by **aggregate-handle**. The DDIF$_EXT aggregate becomes the root aggregate for the external document.

If used, the GET EXTERNAL ENCODING routine must be invoked immediately after an aggregate of type DDIF$_EXT has been returned by the GET AGGREGATE routine. Alternatively, the caller can read the DDIS encoding of an inner document by calling the CDA Toolkit input routines for an inner document root aggregate.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVDOC	Invalid document.
CDA$_INVAGGTYP	Invalid aggregate type.

CDA$GET_STREAM_POSITION
GET STREAM POSITION

Returns the current position in and size of a CDA data stream.

FORMAT

CDA$GET_STREAM_POSITION
stream-handle ,position-rtn ,position-prm ,stream-position ,stream-size

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

stream-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the stream. The **stream-handle** argument is the address of an unsigned longword that contains this stream handle. The handle is returned by a call to either the OPEN STREAM routine or the OPEN FILE routine.

position-rtn

VMS usage: **procedure**
type: **procedure entry mask**
access: **call after stack unwind**
mechanism: **by reference**

Address of a *get-position* routine. The **position-rtn** argument is the address of a procedure entry mask for this *get-position* routine. The calling sequence for a *get-position* routine is defined in the Description section. If you specify 0 for this argument, the CDA Toolkit provides a default *get-position* routine. If you specify a value other than the default for this parameter, you must also specify a value for the **position-prm** argument.

position-prm

VMS usage: **context**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

User context to be passed to the *get-position* routine. The **position-prm** argument contains the value of this user context. This argument should contain the value of the **get-prm** argument passed in a call to the OPEN STREAM or CREATE STREAM routine, or the value of the file handle

in a call to the OPEN FILE or CREATE FILE routine. If you specify a value for the **position-rtn** argument, you must also specify a value for this argument.

stream-position

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the current position (in bytes) as measured from the start of the input stream being processed. The **stream-position** argument is the address of an unsigned longword that receives this position.

stream-size

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the total size (in bytes) of the input stream being processed. The **stream-size** argument is the address of an unsigned longword that receives this size.

DESCRIPTION

The GET STREAM POSITION routine returns the current position and total size of the CDA data stream being processed.

Call Format for User *Get-Position* Routines

The **position-rtn** and **position-prm** arguments are used to invoke a user stream *get-position* routine, and to supply an argument to that routine. This routine returns the size and position of the current data stream.

The call format for a user *get-position* routine is as follows:

position-rtn position-prm ,stream-position ,stream-size

position-prm

VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

User context information that is passed to the GET STREAM POSITION routine. The **position-prm** argument is the address of an unsigned longword that contains this user context.

stream-position

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the current position (in bytes) as measured from the start of the input stream being processed. The **stream-position** argument is the address of an unsigned longword that receives this position.

stream-size

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the total size (in bytes) of the input stream being processed. The **stream-size** argument is the address of an unsigned longword that receives this size.

This user routine must return a completion status. The VMS convention for completion codes is followed: if the low bit of the return value is clear, an error has occurred and the caller returns control to its caller; if the low bit of the return value is set, the caller continues execution.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.

CDA$GET_TEXT_POSITION GET TEXT POSITION

Returns the current position in and size of a text file.

FORMAT

CDA$GET_TEXT_POSITION *file-handle ,file-position ,file-size*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

file-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the text file being processed. The **file-handle** argument is the address of an unsigned longword that contains this file handle. This handle is returned by a call to the OPEN TEXT FILE routine.

file-position

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the current position (in bytes) as measured from the start of the input text file being processed. The **file-position** argument is the address of an unsigned longword that receives this position.

file-size

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the total size (in bytes) of the text file being processed. The **file-size** argument is the address of an unsigned longword that receives this file size.

DESCRIPTION

The GET TEXT POSITION routine returns the current position in and total size of an input text file being processed.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.

CDA$INSERT_AGGREGATE INSERT AGGREGATE

Inserts an aggregate into a sequence. The location at which the aggregate is to be inserted is determined by specifying the preceding aggregate in the sequence.

FORMAT

CDA$INSERT_AGGREGATE *aggregate-handle*
,prev-aggregate-handle

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the aggregate to be inserted into the sequence. The **aggregate-handle** argument is the address of an unsigned longword that contains the handle of the aggregate to be inserted.

prev-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the aggregate after which the aggregate identified by **aggregate-handle** is to be inserted in the sequence. The **prev-aggregate-handle** argument is the address of an unsigned longword that contains this handle.

DESCRIPTION

The INSERT AGGREGATE routine inserts an aggregate into a sequence. The location at which the aggregate is to be inserted is indicated by specifying the preceding aggregate in the sequence.

If the aggregate indicated by **aggregate-handle** is the first aggregate in its own sequence, this entire sequence is inserted into the sequence containing the aggregate specified by **prev-aggregate-handle**. If the aggregate specified as **aggregate-handle** is part of a sequence but is not the first aggregate in that sequence, or if it is the value of an item, an error is returned.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVINSERT	Aggregate already in a sequence.

EXAMPLE

```
    .
    .
    .
aggregate_type = DDIF$_PTH;
status = cda$create_aggregate(&root_aggregate_handle,&aggregate_type,
                              &inner_aggregate_handle);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_SGA_FRM_OUTLINE;
item_length = 4;
status = cda$store_item(&root_aggregate_handle, &aggregate_handle,
                        &aggregate_item,&item_length,
                        &inner_aggregate_handle);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_PTH_C;
local_length = sizeof(integer_value);
integer_value = DDIF$K_PATH_REFERENCE;
status = cda$store_item(&root_aggregate_handle,
                        &inner_aggregate_handle, &aggregate_item,
                        &local_length, &integer_value);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_PTH_REFERENCE;
local_length = sizeof(integer_value);
integer_value = 1;
status = cda$store_item(&root_aggregate_handle,
                        &inner_aggregate_handle, &aggregate_item,
                        &local_length, &integer_value);
if (FAILURE(status)) return(status);

aggregate_type = DDIF$_PTH;
status = cda$create_aggregate(&root_aggregate_handle,
                        &aggregate_type,&inner_aggregate_handle_2);
if (FAILURE(status)) return(status);

status = cda$insert_aggregate(&inner_aggregate_handle_2,
                              &inner_aggregate_handle);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_PTH_C;
local_length = sizeof(integer_value);
integer_value = DDIF$K_PATH_BEZIER;
status = cda$store_item(&root_aggregate_handle,
                        &inner_aggregate_handle_2, &aggregate_item,
                        &local_length, &integer_value);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_PTH_BEZ_PATH_C;
local_length = sizeof(integer_value);
integer_value = DDIF$K_VALUE_CONSTANT;
aggregate_index = 0;
status = cda$store_item(&root_aggregate_handle,
                        &inner_aggregate_handle_2,
                        &aggregate_item, &local_length,
                        &integer_value, &aggregate_index);
if (FAILURE(status)) return(status);
```

```
aggregate_item = DDIF$_PTH_BEZ_PATH;
local_length = sizeof(integer_value);
integer_value = 20;
aggregate_index = 0;
status = cda$store_item(&root_aggregate_handle,
                        &inner_aggregate_handle_2, &aggregate_item,
                        &local_length, &integer_value,
                        &aggregate_index);
if (FAILURE(status)) return(status);

   .
   .
   .
```

This example illustrates the use of the INSERT AGGREGATE routine to insert an aggregate into a sequence.

CDA$LEAVE_SCOPE LEAVE SCOPE

Completes a document that was incrementally written.

FORMAT

CDA$LEAVE_SCOPE *root-aggregate-handle ,stream-handle ,scope-code*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate associated with the document being incrementally written. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

stream-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the stream associated with the document being incrementally written. The **stream-handle** argument is the address of an unsigned longword that contains this stream handle. This handle is returned by a call to either the CREATE FILE routine or the CREATE STREAM routine.

scope-code

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Symbolic constant identifying the scope to be completed. The **scope-code** argument is the address of an unsigned longword containing this code. Valid values are as follows:

Code	Meaning
DDIF$K_DOCUMENT_SCOPE	Document scope
DDIF$K_CONTENT_SCOPE	Content scope
DDIF$K_SEGMENT_SCOPE	Segment scope

DESCRIPTION

The LEAVE SCOPE routine completes a compound document that was incrementally written. For more information on incremental writing of documents, see the description for the ENTER SCOPE routine.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVSCOCOD	Invalid scope code.

Any errors returned by the file routines.

CDA$LOCATE_ITEM LOCATE ITEM

Locates an item within an aggregate by returning its address.

FORMAT

CDA$LOCATE_ITEM *root-aggregate-handle ,aggregate-handle ,aggregate-item ,item-address ,item-length [,aggregate-index] [,add-info]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate with which the aggregate containing the item to be located is associated. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This identifier is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

You must use identical memory management procedures when storing and locating an item within an aggregate, to ensure consistent treatment of memory allocation and deallocation.

aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the aggregate containing the item to be located. The **aggregate-handle** argument is the address of an unsigned longword that contains this aggregate handle.

aggregate-item

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifying code of the item, expressed as a symbolic constant. The **aggregate-item** argument is the address of an unsigned longword that contains this code. The DDIF aggregate item symbolic constants are

defined in the file DDIF$DEF.SDL and are described in Chapter 6 and Appendix D.

The aggregate item DDIF$_USER_CONTEXT is defined for every aggregate type. It is a longword that can be used by the application for any purpose. If you specify this item, the value of **aggregate-item** is initially 0.

For the purpose of this routine, a DDIF$_AGGREGATE_TYPE item is defined for every DDIF aggregate type. If you specify this aggregate item, it returns the type of the aggregate.

item-address

VMS usage: **address**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the address of the item's value. The **item-address** argument is the address of an unsigned longword that receives the address of the value of this item. This storage area can only be read by the calling program; that is, it is read-only. The returned **item-address** is valid until either the Store Item or the ERASE ITEM routine is called for any item in the aggregate, or until the aggregate is deleted.

item-length

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the length (in bytes) of the item's value. The **item-length** argument is the address of an unsigned longword that receives this length.

aggregate-index

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Index of the item (relative to 0). The **aggregate-index** argument is the address of an unsigned longword that contains this index. This argument is required whenever the notation "Array of" appears in the data type of the specified item handle. Otherwise, this argument is only required if the **add-info** argument is also required.

add-info

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives a data type-specific modifier for the data types "Character string" and "string with *add-info*". The **add-info** argument is the address of an unsigned longword that receives this data type-specific information. For data types other than "Character string" and "string with *add-info*", this argument is not written and may be omitted.

For the data type "Character string", the **add-info** parameter receives the character set designator. For the data type "string with *add-info*", if the string value is equal to one of the standard tag values, the **add-info** parameter receives a value that identifies the tag. Otherwise, **add-info** receives a value that indicates that the tag is private.

DESCRIPTION

The LOCATE ITEM routine determines the address of an item within an aggregate. If the located item is encoded as an "Array of", the user must call the GET ARRAY SIZE routine to determine the array size, and then use the LOCATE ITEM routine to read each item in the array by incrementing the **aggregate-index** argument.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVAGGTYP	Invalid aggregate type.
CDA$_INVITMCOD	Invalid item code.
CDA$_EMPTY	Item is empty.
CDA$_INDEX	Index exceeds array bounds.
CDA$_VAREMPTY	Variant item is empty.
CDA$_VARINDEX	Variant index exceeds bounds.
CDA$_VARVALUE	Variant value is undefined.
CDA$_DEFAULT	Value returned is either a default value that is not in the data stream or is an inherited value if inheritance is enabled for the root aggregate.

CDA$NEXT_AGGREGATE NEXT AGGREGATE

Locates the next aggregate in an aggregate sequence.

FORMAT

CDA$NEXT_AGGREGATE *aggregate-handle*
,next-aggregate-handle

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the aggregate to be used in locating the next aggregate. The **aggregate-handle** argument is the address of an unsigned longword containing the handle of the aggregate that precedes the desired aggregate.

next-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the handle of the aggregate that follows the aggregate specified by **aggregate-handle**. The **next-aggregate-handle** argument is the address of an unsigned longword that receives the handle of this aggregate. If the aggregate specified by **aggregate-handle** is the last aggregate in the sequence, **next-aggregate-handle** receives a value of 0.

DESCRIPTION

The NEXT AGGREGATE routine locates the next aggregate in a sequence of aggregates. This aggregate is located by searching for the preceding aggregate, specified by **aggregate-handle**. When using this routine, you should first use the LOCATE ITEM routine to retrieve the aggregate handle of the first aggregate in the sequence. After that, you can use the NEXT AGGREGATE routine to retrieve each additional aggregate in the sequence. All aggregates in the sequence have been retrieved when the status CDA$_ENDOFSEQ is returned.

For example, the DDIF$_CRF_TRANSFORM item in the DDIF$_CRF aggregate is encoded as a sequence of DDIF$_TRN aggregates. After using the LOCATE ITEM routine to retrieve the aggregate handle of the first DDIF$_TRN aggregate in the sequence, you should use the NEXT

AGGREGATE routine to return each additional aggregate in this encoded sequence, until the status CDA$_ENDOFSEQ is returned.

If you are interested in retrieving aggregates from a particular input stream that are not encoded as a sequence, refer to the description of the GET AGGREGATE routine.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_ENDOFSEQ	No successor aggregate found.

CDA$OBJECT_ID_TO_AGGREGATE_TYPE OBJECT ID TO AGGREGATE TYPE

Translates an object identifier to a root aggregate type.

FORMAT

CDA$OBJECT_ID_TO_AGGREGATE_TYPE
buf-len ,buf-adr ,nam-len ,nam-adr ,act-nam-len ,aggregate-type

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

buf-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Length (in bytes) of the object identifier buffer. The **buf-len** argument is the address of an unsigned longword that contains this buffer length.

buf-adr

VMS usage: **vector_longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference, array reference**

Address of the object identifier. The **buf-adr** argument is the address of an array of unsigned longwords that make up the buffer.

nam-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Length (in bytes) of the domain name buffer. The **nam-len** argument is the address of an unsigned longword that contains the length of this domain name buffer.

nam-adr

VMS usage: **vector_longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference, array reference**

Receives the address of the domain name buffer. The **nam-adr** argument is the address of an array of unsigned longwords that comprise the domain name buffer.

act-nam-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the actual length (in bytes) of the domain name in the **nam-adr** buffer. The **act-nam-len** argument is the address of an unsigned longword that receives this actual length.

aggregate-type

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the translated aggregate type. The **aggregate-type** argument is the address of an unsigned longword that receives the aggregate type.

DESCRIPTION

The OBJECT ID TO AGGREGATE TYPE routine translates an object identifier to a root aggregate type.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVAGGTYP	Invalid aggregate type.

CDA$OPEN_CONVERTER OPEN CONVERTER

Activates a front end to process nested content, which can be in the same format as the current document or in a different format.

FORMAT

CDA$OPEN_CONVERTER *standard-item-list*
,converter-context
,front-end-handle

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

standard-item-list

VMS usage: **item_list_2**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference, array reference**

An item list that identifies the document source and destination, and can also contain options to control processing. The **standard-item-list** argument is the address of this item list.

Each entry in the item list is a 2-longword structure with the following format:

item code	buffer length	0
buffer address		4

To terminate the item list, you must specify the final entry or longword as 0. Valid code values for the items in the **standard-item-list** are as follows:

CDA$_INPUT_FORMAT
The parameter is the address and length of a string that specifies the input document format.

CDA$_INPUT_FRONT_END_PROCEDURE
The parameter is the address of the front end's main entry point, DDIF$READ_*format*. The item list length field must be 0. This item enables a caller to provide a front end that is part of the calling application rather than a separate image. If this item code is used, the CDA$_INPUT_FILE item can be used to pass any information (not necessarily a file specification) to the front end.

CDA$_INPUT_FILE
The parameter is the address and length of the file specification of the input document.

CDA$_INPUT_DEFAULT
The parameter is the address and length of the default file specification of the input document. If this parameter is omitted, the front end must supply an appropriate backup default file specification.

CDA$_INPUT_PROCEDURE
The parameter is the address of a procedure to provide input. The item list length field must be 0. The input procedure must conform to the requirements for a *get* routine. The calling sequence for a user *get* routine is defined in the Description section of this routine.

CDA$_INPUT_PROCEDURE_PARM
The parameter is the address of a longword parameter to the input procedure. The item list length field must be 4.

CDA$_INPUT_POSITION_PROCEDURE
The parameter is the address of a procedure that provides position information. The item list length field must be set to 0.

CDA$_INPUT_ROOT_AGGREGATE
The parameter is the address of a longword handle to a root aggregate that specifies an in-memory input document. The item list length field must be 4. The in-memory structure, except for the root aggregate itself, is erased by this operation. The root aggregate must specify standard memory allocation.

converter-context

VMS usage: **context**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Context value passed as a parameter to the DDIF$READ_*format* entry point in the front end. The **converter-context** argument is the address of an unsigned longword containing this context.

front-end-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the handle of the front end that will process the nested content. The **front-end-handle** argument is the address of an unsigned longword that receives this front end handle. This handle must be used in all subsequent operations relating to that front end.

DESCRIPTION

The OPEN CONVERTER routine activates a front end to process nested content, which may be in the same format as the current document or in a different format. Processing options that were specified to the CONVERT DOCUMENT routine for the document format are retrieved and appended to the item list.

Call Format for User *Get* Routines

The **get-rtn** and **get-prm** arguments are used to invoke a user stream *get* routine, and to supply an argument to that routine. The call format for this user routine is as follows:

get-rtn get-prm ,num-bytes ,buf-adr

get-prm
VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

User context argument. The **get-prm** argument contains the value of the parameter to be passed to the user *get* routine.

num-bytes
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the number of bytes contained in the buffer. The **num-bytes** argument is the address of an unsigned longword that receives this number. The number of bytes is zero if and only if the stream does not contain any more data.

buf-adr
VMS usage: **address**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the address of the buffer. The **buf-adr** argument is the address of an unsigned longword that receives the buffer address.

This user routine must return a completion status. The VMS convention for completion codes is followed: if the low bit of the return value is clear, an error has occurred and the caller returns control to its caller; if the low bit of the return value is set, the caller continues execution.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_UNSUPFMT	Unsupported document format.

Any error returned by the specific front end.

CDA$OPEN_FILE OPEN FILE

Opens the specified file for input and validates that its contents are valid compound document data. An input stream and a root aggregate are also created.

FORMAT

CDA$OPEN_FILE *file-spec-len ,file-spec default-file-spec-len ,default-file-spec ,alloc-rtn ,dealloc-rtn ,alloc-dealloc-prm ,aggregate-type ,processing-options ,result-file-spec-len ,result-file-spec ,result-file-ret-len ,stream-handle ,file-handle ,root-aggregate-handle*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

file-spec-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The length of the string specified by the **file-spec** parameter. The **file-spec-len** argument is the address of an unsigned longword that contains this file specification length.

file-spec

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by reference**

The file specification. The **file-spec** argument is the address of a character string that contains this file specification.

default-file-spec-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The length (in bytes) of the buffer specified by **default-file-spec**. The **default-file-spec-len** argument is the address of an unsigned longword

that contains this buffer length. If you specify an address of 0 for both the **default-file-spec-len** and **default-file-spec** arguments, a default file specification of ".DDIF" is used.

default-file-spec

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by reference**

The default file specification. The **default-file-spec** argument is the address of a character string that contains the default file specification. In order to simplify the porting of applications, the character string should consist of only a file type in lowercase characters. If you specify an address of 0 for both the **default-file-spec-len** and **default-file-spec** arguments, a default file specification of ".DDIF" is used.

alloc-rtn

VMS usage: **procedure**
type: **procedure entry mask**
access: **call after stack unwind**
mechanism: **by reference**

Address of a memory allocation routine. The **alloc-rtn** argument is the address of a procedure entry mask for this allocation routine. The calling sequence for an allocation routine is defined in the Description section of this routine. On VMS systems, if you specify 0 for this argument, LIB$GET_VM is used as the memory allocation routine.

dealloc-rtn

VMS usage: **procedure**
type: **procedure entry mask**
access: **call after stack unwind**
mechanism: **by reference**

Address of a memory deallocation routine. The **dealloc-rtn** argument is the address of a procedure entry mask for this deallocation routine. The calling sequence for a deallocation routine is defined in the Description section of this routine. On VMS systems, if you specify 0 for this argument, LIB$FREE_VM is used as the memory deallocation routine.

alloc-dealloc-prm

VMS usage: **context**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

User context to be passed to the memory allocation and deallocation routines. The **alloc-dealloc-prm** argument contains the value of this user context. If the VMS system default memory allocation or deallocation routine is used, this value must be a zone identifier or 0 for the default zone.

aggregate-type

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The type of aggregate, expressed as a symbolic constant. The **aggregate-type** argument is the address of an unsigned longword that contains this symbolic constant. The only valid root aggregate type is DDIF$_DDF.

processing-options

VMS usage: **item_list_2**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference, array reference**

An item list containing options to control processing. The **processing-options** argument is the address of this item list. Each entry in the item list is a 2-longword structure; to terminate the item list you must specify a final entry or longword of zero. Valid item codes are as follows:

DDIF$_INHERIT_ATTRIBUTES	If a style guide is specified in the document header, definitions in the style guide are appended to the definitions present on the root segment, provided they are not hidden by definitions in the document.
DDIF$_RETAIN_DEFINITIONS	Segment definitions that enable the operation of CDA$FIND_DEFINITION are retained. This item code is required only if neither DDIF$_INHERIT_ATTRIBUTES nor DDIF$_EVALUATE_CONTENT is specified.
DDIF$_EVALUATE_CONTENT	If a content reference is external, the content is fetched from the external document provided it is either remote content or copy content that is not present in the document.
DDIF$_DISCARD_I_SEGMENTS	Segments of the image ($I) content category, and any nested segments, are discarded.
DDIF$_DISCARD_2D_SEGMENTS	Segments of the graphics ($2D) content category, and any nested segments, are discarded.
DDIF$_DISCARD_T_SEGMENTS	Segments of the text ($T) content category, and any nested segments, are discarded.
DDIF$_DISCARD_TBL_SEGMENTS	Segments of the table ($TBL) content category, and any nested segments, are discarded.
DDIF$_DISCARD_PDL_SEGMENTS	Segments of the page description language ($PDL) content category, and any nested segments, are discarded.

result-file-spec-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Length of the buffer (in bytes) specified by **result-file-spec**. The **result-file-spec-len** argument is the address of an unsigned longword containing this length. If you specify 0 for this parameter, the resultant file specification length is not returned.

result-file-spec

VMS usage: **char_string**
type: **character string**
access: **write only**
mechanism: **by reference**

Receives the resultant file specification. The **result-file-spec** argument is the address of a character string that receives this file specification. If you specify 0 for this parameter, the resultant file specification is not returned. This file specification is the result of a VMS RMS $OPEN operation.

result-file-ret-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the actual length (in bytes) of the resultant file specification. The **result-file-ret-len** argument is the address of an unsigned longword that receives the actual length of the resultant file specification. If you specify 0 for this parameter, the actual length of the resultant file specification is not returned.

stream-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives a value that identifies the newly created stream. The **stream-handle** argument is the address of an unsigned longword that receives this stream handle. This handle must be used in all subsequent operations on that stream.

file-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives a value that identifies the newly opened file. The **file-handle** argument is the address of an unsigned longword that receives this file handle. This handle must be used in all subsequent operations on that file.

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives a value that identifies the newly created root aggregate. The **root-aggregate-handle** argument is the address of an unsigned longword that receives this root aggregate handle. This handle must be used in all subsequent operations on that root aggregate.

DESCRIPTION

The OPEN FILE routine opens a file for input and validates that the contents of the file are compound document data. At the same time, this routine also creates an input stream and a root aggregate.

Call Format for User Allocation/Deallocation Routines

The **alloc-rtn**, **dealloc-rtn**, and **alloc-dealloc-prm** arguments are used to invoke a user routine that performs memory allocation or deallocation, and to supply an argument to that routine. The call format for one of these user routines is as follows:

user-rtn num-bytes ,base-adr ,alloc-dealloc-prm

num-bytes
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**
The number of bytes to allocate or free. The **num-bytes** argument is the address of an unsigned longword that contains this number of bytes. The value of **num-bytes** must be greater than zero.

base-adr
VMS usage: **address**
type: **longword (unsigned)**
access: **read only or write only**
mechanism: **by reference**
Virtual address of the first byte of memory allocated or freed. The **base-adr** argument is the address of an unsigned longword containing this base address. (This argument is write-only for a *get* routine, and read-only for a *free* routine.)

alloc-dealloc-prm
VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**
User context argument. The **alloc-dealloc-prm** argument contains the value of the parameter to be passed to the user routine.

Each of these user routines must return a completion status. The VMS convention for completion codes is followed: if the low bit of the return value is clear, an error has occurred and the caller returns control to its caller; if the low bit of the return value is set, the caller continues execution.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVAGGTYP	Invalid aggregate type.
CDA$_INVITMLST	Invalid item list.

Any error returned by the memory allocation routines.

Any error returned by the file routines.

EXAMPLE

```
    .
    .
    .
/* Open the file for input */

aggregate_type = DDIF$_DDF;
status = cda$open_file(&filename_length,
                       &test1_filename[0],
                       0,
                       0,
                       0,
                       0,
                       0,
                       &aggregate_type,
                       0,
                       &result_file_spec_len,
                       &result_file_spec[0],
                       &result_file_ret_len,
                       &stream_handle,
                       &file_handle,
                       &root_aggregate_handle );
if (FAILURE(status)) return(status);

/* Read the entire document in, then close the file */
printf("Reading document...\n");
status = cda$get_document(&root_aggregate_handle, &stream_handle);
if (FAILURE(status)) return(status);

status = cda$close_file(&stream_handle, &file_handle);
if (FAILURE(status)) return(status);

    .
    .
    .
```

This example illustrates a typical call to the OPEN FILE routine. Following a call to this routine, the file is read using the GET DOCUMENT routine and subsequently closed.

CDA$OPEN_STREAM OPEN STREAM

Opens a compound document stream for input.

FORMAT

CDA$OPEN_STREAM *alloc-rtn ,dealloc-rtn ,alloc-dealloc-prm ,get-rtn ,get-prm ,stream-handle*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

alloc-rtn

VMS usage: **procedure**
type: **procedure entry mask**
access: **call after stack unwind**
mechanism: **by reference**

Address of a memory allocation routine. The **alloc-rtn** argument is the address of a procedure entry mask for this allocation routine. The calling sequence for an allocation routine is defined in the Description section of this routine. On VMS systems, if you specify 0 for this argument, LIB$GET_VM is used as the memory allocation routine.

dealloc-rtn

VMS usage: **procedure**
type: **procedure entry mask**
access: **call after stack unwind**
mechanism: **by reference**

Address of a memory deallocation routine. The **dealloc-rtn** argument is the address of a procedure entry mask for this deallocation routine. The calling sequence for a deallocation routine is defined in the Description section of this routine. On VMS systems, if you specify 0 for this argument, LIB$FREE_VM is used as the memory deallocation routine.

alloc-dealloc-prm

VMS usage: **context**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

User context to be passed to the memory allocation and deallocation routines. The **alloc-dealloc-prm** argument contains the value of this user context. If the VMS system default memory allocation or deallocation procedure is used, this value must be a zone identifier or 0 for the default zone.

get-rtn

VMS usage: **procedure**
type: **procedure entry mask**
access: **call after stack unwind**
mechanism: **by reference**

Address of a stream *get* routine. The **get-rtn** argument is the address of a procedure entry mask for this stream *get* routine. The calling sequence for a *get* routine is defined in the Description section of this routine. If you specify 0 for this argument, the VMS RMS $GET service is used. If you specify a value other than the default for this argument, you must also specify a value for the **get-prm** argument.

get-prm

VMS usage: **context**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

User context to be passed to the stream *get* routine. The **get-prm** argument contains the value of this user context. If the VMS system default *get* routine is used, the value must be a pointer to a RAB.

stream-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives a value that identifies the newly created stream. The **stream-handle** argument is the address of an unsigned longword that receives this stream handle. This handle must be used in all subsequent operations on that stream.

DESCRIPTION

The OPEN STREAM routine opens a compound document stream for input. The number of streams that you can open simultaneously is limited only by the amount of memory available.

Call Format for User Allocation/Deallocation Routines

The **alloc-rtn**, **dealloc-rtn**, and **alloc-dealloc-prm** arguments are used to invoke a user routine that performs memory allocation or deallocation, and to supply an argument to that routine. The call format for one of these user routines is as follows:

user-rtn num-bytes ,base-adr ,alloc-dealloc-prm

num-bytes

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The number of bytes to allocate or free. The **num-bytes** argument is the address of an unsigned longword that contains this number of bytes. The value of **num-bytes** must be greater than 0.

base-adr
VMS usage: **address**
type: **longword (unsigned)**
access: **read only or write only**
mechanism: **by reference**
Virtual address of the first byte of memory allocated or freed. The **base-adr** argument is the address of an unsigned longword containing this base address. (This argument is write-only for a *get* routine, and read-only for a *free* routine.)

alloc-dealloc-prm
VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**
User context argument. The **alloc-dealloc-prm** argument contains the value of the parameter to be passed to the user routine.

Call Format for User *Get* Routines

The **get-rtn** and **get-prm** arguments are used to invoke a user stream *get* routine and to supply an argument to that routine. This routine reads bytes from an input stream. The buffer is supplied by a call to the *get* routine; therefore, the application can use any buffer management technique. The caller of the *get* routine treats the buffer as read-only; it must contain valid data until the next call to the *get* routine.

The call format for a user *get* routine is as follows:

get-rtn get-prm ,num-bytes ,buf-adr

get-prm
VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**
User context argument. The **get-prm** argument contains the value of the parameter to be passed to the user *get* routine.

num-bytes
VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**
Receives the number of bytes contained in the buffer. The **num-bytes** argument is the address of an unsigned longword that receives this number. The number of bytes is 0 only if the stream does not contain any more data.

buf-adr
VMS usage: **address**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the address of the buffer. The **buf-adr** argument is the address of an unsigned longword that receives the buffer address.

Each of these user routines must return a completion status. The VMS convention for completion codes is followed: if the low bit of the return value is clear, an error has occurred and the caller returns control to its caller; if the low bit of the return value is set, the caller continues execution.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.

Any error returned by the memory allocation routines.

CDA$OPEN_TEXT_FILE OPEN TEXT FILE

Opens a standard text file for input.

FORMAT

CDA$OPEN_TEXT_FILE *file-spec-len ,file-spec ,default-file-spec-len ,default-file-spec ,result-file-spec-len ,result-file-spec ,result-file-ret-len ,text-file-handle*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

file-spec-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Length (in bytes) of the string specified by the **file-spec** argument. The **file-spec-len** argument is the address of an unsigned longword that contains this length.

file-spec

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by reference**

File specification of the text file to be opened for input. The **file-spec** argument is the address of a character string containing this file specification.

default-file-spec-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Length (in bytes) of the string specified by **default-file-spec**. The **default-file-spec-len** argument is the address of an unsigned longword that contains this default file specification length. If you specify 0 for this parameter, no default file specification is used.

default-file-spec

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by reference**

Default file specification. The **default-file-spec** argument is the address of a character string that contains this default file specification. If you specify a 0 for this parameter, no default file specification is used. The string should consist only of a file type in lowercase characters.

result-file-spec-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Length (in bytes) of the buffer specified by **result-file-spec**. The **result-file-spec-len** argument is the address of an unsigned longword that contains this buffer length. If you specify 0 for this parameter, the length of the resultant file specification is not returned.

result-file-spec

VMS usage: **char_string**
type: **character string**
access: **write only**
mechanism: **by reference**

Receives the resultant file specification. The **result-file-spec** argument is the address of a character string that receives this resultant file specification. This file specification is the result of a VMS RMS $OPEN operation. If you specify 0 for this parameter, a resultant file specification is not returned.

result-file-ret-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the actual length (in bytes) of the resultant file specification. The **result-file-ret-len** argument is the address of an unsigned longword that receives the actual length of the resultant file specification. If you specify 0 for this parameter, the actual length of the resultant file specification is not returned.

text-file-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the handle of the text file. The **text-file-handle** argument is the address of an unsigned longword that contains this text file handle. This handle must be used in all subsequent operations on that text file.

DESCRIPTION

The OPEN TEXT FILE routine opens a standard text file for input.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.

Any error returned by the memory allocation routines.

Any error returned by the file routines.

CDA$PRUNE_AGGREGATE PRUNE AGGREGATE

Removes the next sequential document content aggregate from an existing in-memory compound document, and returns its handle and type.

FORMAT

CDA$PRUNE_AGGREGATE
root-aggregate-handle
,aggregate-handle ,aggregate-type

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate associated with the aggregate to be removed. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This aggregate handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

When removing aggregates using the PRUNE AGGREGATE routine, you must use the same value for the root aggregate handle argument consistently to remove all the aggregates in the compound document. Once you have removed all of the aggregates, you cannot specify the same root aggregate handle again when calling the PRUNE AGGREGATE routine.

aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the handle of the removed aggregate. The **aggregate-handle** argument is the address of an unsigned longword that receives this aggregate handle. This handle must be used in all subsequent operations on that aggregate.

aggregate-type

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the aggregate type. The **aggregate-type** argument is the address of an unsigned longword that receives this aggregate type. If the aggregate type returned is DDIF$_EOS (end of segment), the value of the aggregate handle argument is 0.

DESCRIPTION

The PRUNE AGGREGATE routine removes the next sequential document content aggregate from an existing in-memory compound document, and returns the aggregate identifier and type. A front end should invoke this routine from the *get-aggregate* entry point module in cases where it builds an entire compound document in memory before returning its content.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_ENDOFDOC	End of document.

CDA$PRUNE_POSITION PRUNE POSITION

Returns the position in and size of an in-memory document.

FORMAT

CDA$PRUNE_POSITION *root-aggregate-handle ,file-position ,file-size*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate associated with the in-memory document. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. The handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

file-position

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the current position (in bytes) as measured from the start of the document being processed. The **file-position** argument is the address of an unsigned longword that receives this position.

file-size

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the total size (in bytes) of the in-memory document being processed. The **file-size** argument is the address of an unsigned longword that receives this size.

DESCRIPTION

The PRUNE POSITION routine returns the current position in and total size of the in-memory document being processed. This routine must be used by the *get-position* routine when a front end builds an entire document in memory before returning its content.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.

CDA$PUT_AGGREGATE PUT AGGREGATE

Writes one or more aggregates to a specified stream.

FORMAT

CDA$PUT_AGGREGATE *root-aggregate-handle*
,stream-handle
,aggregate-handle

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate associated with the aggregate to be written. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

When writing aggregates using the PUT AGGREGATE routine, you must use the same value for **root-aggregate-handle** consistently to write all the aggregates in the compound document. Once you have written all of the aggregates, you cannot specify the same **root-aggregate-handle** again when calling this routine.

stream-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the stream to which the aggregate is to be written. The **stream-handle** argument is the address of an unsigned longword that contains this stream handle. This handle is returned by a call to either the CREATE FILE routine or the CREATE STREAM routine.

aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the aggregate to be written. The **aggregate-handle** argument is the address of an unsigned longword that contains this aggregate handle. This handle is returned by a call to any one of the

following routines: CONVERT AGGREGATE, CREATE AGGREGATE, GET AGGREGATE, or LOCATE ITEM.

DESCRIPTION

The PUT AGGREGATE routine writes one or more aggregates to a specified stream. Note that the aggregates remain unchanged after a call to this routine. If you do not require these aggregates after you call this routine, your application should include a subsequent call to the DELETE AGGREGATE routine to destroy these aggregates.

If the aggregate is part of a sequence, a call to the PUT AGGREGATE routine causes the entire sequence to be written. The aggregate type of the written aggregate is one of the following:

Aggregate Type	Meaning
DDIF$_DSC	Document descriptor
DDIF$_DHD	Document header
DDIF$_SEG	Document segment
DDIF$_TXT	Text content
DDIF$_GTX	General text content
DDIF$_HRD	Hard directive
DDIF$_SFT	Soft directive
DDIF$_HRV	Hard value directive
DDIF$_SFV	Soft value directive
DDIF$_BEZ	Bézier curve content
DDIF$_LIN	Polyline content
DDIF$_ARC	Arc content
DDIF$_FAS	Fill area set content
DDIF$_IMG	Image content
DDIF$_CRF	Content reference
DDIF$_EXT	External content
DDIF$_PVT	Private content
DDIF$_GLY	Layout galley
DDIF$_EOS	End of segment

If the aggregate is of type DDIF$_SEG, the segment content must be specified by the value of the DDIF$_SEG_CONTENT item. If the segment does not contain content, you must use the ENTER SCOPE routine to write the segment aggregate. Note that any lower-level content must be attached to the segment aggregate before it is written.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVAGGTYP	Invalid aggregate type.
CDA$_INVDOC	Invalid document content.

Any error returned by the file routines.

CDA$PUT_DOCUMENT PUT DOCUMENT

Writes an entire document to the specified stream. The document is not changed by this operation.

FORMAT

CDA$PUT_DOCUMENT *root-aggregate-handle ,stream-handle*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate associated with the document to be written. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

Once you write an entire document, you cannot call the PUT DOCUMENT routine specifying the same root aggregate handle again. That is, you can only write a document associated with a particular root aggregate once.

stream-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the stream to which the document is to be written. The **stream-handle** argument is the address of an unsigned longword that contains this stream handle. This handle is returned by a call to either the CREATE FILE routine or the CREATE STREAM routine.

DESCRIPTION

The PUT DOCUMENT routine writes an entire document to a specified stream. Note that the document remains unchanged after a call to this routine. If you do not require the in-memory structure after you call this routine, your application should include a subsequent call to the DELETE ROOT AGGREGATE routine to destroy this structure.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVDOC	Invalid document content.

Any error returned by the file routines.

CDA$READ_TEXT_FILE READ TEXT FILE

Reads a line from a standard text file.

FORMAT

CDA$READ_TEXT_FILE *text-file-handle ,buffer-length ,buffer-address*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

text-file-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the text file from which the line is to be read. The **text-file-handle** argument is the address of an unsigned longword that contains this text file handle. This handle is returned by a call to the OPEN TEXT FILE routine.

buffer-length

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the length (in bytes) of the line that is read. The **buffer-length** argument is the address of an unsigned longword that receives this length.

buffer-address

VMS usage: **address**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the address of the line that is read. The **buffer-address** argument is the address of an unsigned longword that receives the address of this line. No trailing record delimiter is present.

DESCRIPTION

The READ TEXT FILE routine reads a line from a standard text file. On VMS systems, the line is the next RMS record in the file.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_ENDOFDOC	End of document.

Any error returned by the file routines.

CDA$REMOVE_AGGREGATE REMOVE AGGREGATE

Removes an aggregate from a sequence. If the specified aggregate is not part of a sequence, no operation is performed.

FORMAT

CDA$REMOVE_AGGREGATE *aggregate-handle*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the aggregate to be removed from the sequence. The **aggregate-handle** argument is the address of an unsigned longword that contains this aggregate handle.

DESCRIPTION

The REMOVE AGGREGATE routine removes an aggregate that is part of a sequence from that sequence. If the specified aggregate is not part of a sequence, no operation is performed.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.

CDA$STORE_ITEM STORE ITEM

Writes the contents of an item within an aggregate. If the item is indexed, the index must not exceed one more than the number of existing items.

FORMAT

CDA$STORE_ITEM *root-aggregate-handle ,aggregate-handle ,aggregate-item ,buf-len ,buf-adr [,aggregate-index] [,add-info]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

root-aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the root aggregate with which the aggregate containing the item is associated. The **root-aggregate-handle** argument is the address of an unsigned longword that contains this root aggregate handle. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

You must use identical memory management procedures when storing and locating an item within an aggregate, to ensure consistent treatment of memory allocation and deallocation.

aggregate-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the aggregate into which the item is written. The **aggregate-handle** argument is the address of an unsigned longword that contains this aggregate handle.

aggregate-item

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifying code of the item, expressed as a symbolic constant. The **aggregate-item** argument is the address of an unsigned longword that contains this code. The DDIF aggregate item symbolic constants are

defined in the file DDIF$DEF.SDL and are described in Chapter 6 and Appendix D.

buf-len

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Length (in bytes) of the buffer specified by the **buf-adr** argument. The **buf-len** argument is the address of an unsigned longword that contains this buffer length.

buf-adr

VMS usage: **vector_byte_unsigned**
type: **byte (unsigned)**
access: **read only**
mechanism: **by reference, array reference**

Buffer containing the item's value. The **buf-adr** argument is the address of an array of unsigned bytes that make up the buffer.

aggregate-index

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Index of the item (relative to 0). The **aggregate-index** argument is the address of an unsigned longword that contains this index. This argument is required whenever the notation "Array of" appears in the data type of the specified item handle. Otherwise, this argument is only required if the **add-info** argument is also required.

add-info

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Data type-specific modifier for the data types "character string" and "string with *add-info*". The optional **add-info** argument is the address of an unsigned longword that contains this data type-specific information. For data types other than "character string" and "string with *add-info*", this argument is ignored and may be omitted.

For the data type "character string", the **add-info** parameter contains the character set designator. For the data type "string with *add-info*", if the string value is equal to one of the standard tag values, the **add-info** parameter contains a value that identifies the tag. Otherwise, **add-info** contains a value that indicates that the tag is private.

DESCRIPTION

The STORE ITEM routine lets you store the value of each item within an aggregate. After creating an aggregate, you must use this routine to fill in the appropriate items in the aggregate. The items that exist for each aggregate are defined in the file DDIF$DEF.SDL and are described in Chapter 6 and Appendix D. Note that there are optional and required aggregate items. If the text does not specify that the item is optional, then it must be specified in order to create a valid aggregate of that type.

If an aggregate item is indexed, the index must not exceed one more than the number of existing items. If the item is of data type variable, the value of the item that determines the data type must have been previously established.

The STORE ITEM routine erases the previous item value, unless the item is "aggregate-valued" and not empty. (An "aggregate-valued" item is one in which the value of the aggregate is actually the handle of another aggregate.) In the case of an item that is aggregate valued and not empty, the specified aggregate is inserted in sequence before the existing aggregate. If the specified aggregate is the beginning of a sequence, the entire sequence is inserted before the existing aggregate. If the specified aggregate is part of a sequence but is not the first aggregate in the sequence, or if the specified aggregate is the value of an item, an error is returned.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.
CDA$_INVAGGTYP	Invalid aggregate type.
CDA$_INVITMCOD	Invalid item code.
CDA$_INDEX	Index exceeds array bounds.
CDA$_VAREMPTY	Variant item is empty.
CDA$_VARINDEX	Variant index exceeds bounds.
CDA$_VARVALUE	Variant value is undefined.
CDA$_INVINSERT	Aggregate already in a sequence.
CDA$_INVBUFLEN	Invalid buffer length.

EXAMPLES

1

```
        .
        .
        .
static unsigned char
    product_name[] = {"Sample Product"};
        .
        .
        .
aggregate_type = DDIF$_DSC;
status = cda$create_aggregate(&root_aggregate_handle,
                              &aggregate_type, &aggregate_handle);
if (FAILURE(status)) return(status);
```

```
aggregate_item = DDIF$_DDF_DESCRIPTOR;
local_length = sizeof(aggregate_handle);
status = cda$store_item(&root_aggregate_handle,
                        &root_aggregate_handle, &aggregate_item,
                        &local_length, &aggregate_handle);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_DSC_MAJOR_VERSION;
local_length = sizeof(integer_value);
integer_value = 1;
status = cda$store_item(&root_aggregate_handle, &aggregate_handle,
                        &aggregate_item, &local_length,
                        &integer_value);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_DSC_MINOR_VERSION;
local_length = sizeof(integer_value);
integer_value = 0;
status = cda$store_item(&root_aggregate_handle, &aggregate_handle,
                        &aggregate_item, &local_length,
                        &integer_value);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_DSC_PRODUCT_IDENTIFIER;
local_length = 7;
status = cda$store_item(&root_aggregate_handle, &aggregate_handle,
                        &aggregate_item, &local_length, "Example");
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_DSC_PRODUCT_NAME;
local_len = sizeof(product_name);
aggregate_index = 0;
add_info = CDA$K_ISO_LATIN1;
status = cda$store_item(&root_aggregate_handle, &aggregate_handle,
                        &aggregate_item, &local_length,
                        product_name, &aggregate_index,
                        &add_info);
if (FAILURE(status)) return(status);

     .
     .
     .
```

This example illustrates the creation of a document descriptor aggregate (type DDIF$_DSC), and the use of the STORE ITEM routine to fill in the items in the aggregate.

2

```
     .
     .
     .
aggregate_type = DDIF$_TRN;
status = cda$create_aggregate(&root_aggregate_handle,
                              &aggregate_type, &inner_aggregate_handle);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_SGA_FRM_TRANSFORM;
item_length = 4;
status = cda$store_item(&root_aggregate_handle,
                        &aggregate_handle, &aggregate_item,
                        &item_length, &inner_aggregate_handle);
if (FAILURE(status)) return(status);
```

```
aggregate_item = DDIF$_TRN_PARAMETER_C;
local_length = sizeof(integer_value);
integer_value = DDIF$K_X_SCALE;
status = cda$store_item(&root_aggregate_handle,
                        &inner_aggregate_handle, &aggregate_item,
                        &local_length, &integer_value);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_TRN_PARAMETER;
local_length = sizeof(float_value);
float_value = 3.5;
status = cda$store_item(&root_aggregate_handle,
                        &inner_aggregate_handle, &aggregate_item,
                        &local_length, &float_value);
if (FAILURE(status)) return(status);

aggregate_type = DDIF$_TRN;
status = cda$create_aggregate(&root_aggregate_handle,
                        &aggregate_type, &inner_aggregate_handle_2);
if (FAILURE(status)) return(status);

status = cda$insert_aggregate(&inner_aggregate_handle_2,
                              &inner_aggregate_handle);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_TRN_PARAMETER_C;
local_length = sizeof(integer_value);
integer_value = DDIF$K_MATRIX_2_BY_3;
status = cda$store_item(&root_aggregate_handle,
                        &inner_aggregate_handle_2, &aggregate_item,
                        &local_length, &integer_value);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_TRN_PARAMETER;
local_length = sizeof(float_value);
float_value = 4.75;
aggregate_index = 0;
status = cda$store_item(&root_aggregate_handle,
                        &inner_aggregate_handle_2, &aggregate_item,
                        &local_length, &float_value,
                        &aggregate_index);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_TRN_PARAMETER;
local_length = sizeof(float_value);
float_value = 6.11;
aggregate_index = 1;
status = cda$store_item(&root_aggregate_handle,
                        &inner_aggregate_handle_2, &aggregate_item,
                        &local_length, &float_value,
                        &aggregate_index);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_TRN_PARAMETER;
local_length = sizeof(float_value);
float_value = 2.22;
aggregate_index = 2;
status = cda$store_item(&root_aggregate_handle,
                        &inner_aggregate_handle_2, &aggregate_item,
                        &local_length, &float_value,
                        &aggregate_index);
if (FAILURE(status)) return(status);
```

```
aggregate_item = DDIF$_TRN_PARAMETER;
local_length = sizeof(float_value);
float_value = 3.0;
aggregate_index = 3;
status = cda$store_item(&root_aggregate_handle,
                        &inner_aggregate_handle_2, &aggregate_item,
                        &local_length, &float_value,
                        &aggregate_index);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_TRN_PARAMETER;
local_length = sizeof(float_value);
float_value = 1.25;
aggregate_index = 4;
status = cda$store_item(&root_aggregate_handle,
                        &inner_aggregate_handle_2, &aggregate_item,
                        &local_length, &float_value,
                        &aggregate_index);
if (FAILURE(status)) return(status);

aggregate_item = DDIF$_TRN_PARAMETER;
local_length = sizeof(float_value);
float_value = 2.15;
aggregate_index = 5;
status = cda$store_item(&root_aggregate_handle,
                        &inner_aggregate_handle_2, &aggregate_item,
                        &local_length, &float_value,
                        &aggregate_index);
if (FAILURE(status)) return(status);

     .
     .
     .
```

This example illustrates the use of the STORE ITEM routine to specify two transformation aggregates (type DDIF$_TRN). The type of transformation specified by the DDIF$_TRN aggregate is indicated by the value of the DDIF$_TRN_PARAMETER_C item. The first transformation aggregate specifies an *x*-scale transformation. The second transformation aggregate specifies a 2 x 3 matrix transformation of the following format:

A	D	0
B	E	0
C	F	1

Each matrix coefficient is stored in the DDIF$_TRN aggregate in each call to the STORE ITEM routine. The first call to STORE ITEM for this matrix writes the A matrix coefficient into array item 0; the second call writes B to array item 1, and so on until coefficients A through F are written to the array. You are responsible for updating the aggregate index of the array each time a coefficient is written. One matrix coefficient is stored in each call to the STORE ITEM routine. The aggregate index is used to specify which matrix coefficient is being written.

CDA$WRITE_TEXT_FILE WRITE TEXT FILE

Writes a line of text to a standard text file.

FORMAT

CDA$WRITE_TEXT_FILE *text-file-handle*
,buffer-length
,buffer-address

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

text-file-handle

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the text file to which the line is written. The **text-file-handle** argument is the address of an unsigned longword that contains this text file handle. This handle is returned by a call to the CREATE TEXT FILE routine.

buffer-length

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Length (in bytes) of the buffer specified by the **buffer-address** argument. The **buffer-length** argument is the address of an unsigned longword that contains this buffer length.

buffer-address

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by reference**

The line to be written to the text file. The **buffer-address** argument is the address of a character string containing this line.

DESCRIPTION

The WRITE TEXT FILE routine writes a line of text to a standard text file. On VMS systems, the written line becomes an RMS record.

CONDITION VALUES RETURNED

CDA$_NORMAL	Normal successful completion.

Any error returned by the file routines.

A VMS Support for CDA in DECwindows

VMS commands and utilities, as well as existing application programs that accept text input, can now use the text content of DECwindows compound documents.

To support the use of DDIF text, VMS RMS has implemented a new RMS file attribute, **stored semantics,** and a DDIF-to-Text **RMS extension**. The value of the stored semantics attribute is called the file **tag**; it specifies how file data is to be interpreted. When file data is to be interpreted in accordance with the DDIF specification, the appropriate file tag is DDIF. The use of file tags is limited to disk files on VMS DECwindows systems.

The DDIF-to-Text RMS extension transparently extracts text from DDIF files as variable-length text records that can be accessed through the VMS RMS interface.

The enhancements made to support the reading of text from DDIF files are transparent to the user and to the application programmer. This support requires that all DDIF files in a VMS DECwindows environment be tagged with the DDIF file tag. DDIF files created by VMS DECwindows software are tagged appropriately.

Section A.1 describes various VMS file management commands and utilities that display, create and preserve file tags where appropriate. Section A.1 also describes the way various VMS commands and utilities respond to DDIF file input. Section A.2 describes VMS support for DDIF files in heterogeneous computing environments. Section A.3 describes the changes made to the VMS RMS program interface to support the stored semantics attribute and to control access to the content of DDIF files.

A.1 VMS Commands and Utilities

This section describes the VMS commands and utilities that support tag maintenance by displaying, creating and preserving the RMS file tags used with DDIF files. It also provides additional information that is relevant to the way selected VMS commands and utilities respond to DDIF file input.

The following table lists the VMS commands and utilities that support tag maintenance.

Command/Utility	Tag Maintenance Function
DIRECTORY/FULL	Displays file tag
ANALYZE/RMS_FILE	Displays file tag
SET FILE/SEMANTICS	Creates file tag

Command/Utility	Tag Maintenance Function
VMS MAIL	Preserves file tag†
COPY	Preserves file tag†
BACKUP	Preserves file tag

†See text for exceptions.

Tags are made up of binary values that can be up to 64 bytes long and can be expressed using hexadecimal notation. The hexadecimal value of the DDIF tag, for example, is 2B0C8773010301. VMS permits you to assign mnemonics to tag values so that DCL commands like DIRECTORY/FULL and VMS utilities like FDL and ANALYZE/RMS_FILE display a mnemonic for the DDIF tag instead of the hexadecimal value. The following DCL commands have been included in the system startup command file to assign the mnemonic DDIF to the hexadecimal value for a DDIF tag.

```
$ DEFINE/TABLE=RMS$SEMANTIC_TAGS DDIF 2B0C8773010301
$ DEFINE/TABLE=RMS$SEMANTIC_OBJECTS "2B0C8773010301" DDIF
```

Using the appropriate DEFINE commands, you can assign mnemonics for other tags, including tags used with international program applications.

A.1.1 Displaying RMS File Tags

The DIRECTORY/FULL command and the Analyze/RMS_File Utility now display the RMS file tag for DDIF files.

A.1.1.1 DIRECTORY/FULL

Where applicable, the DIRECTORY/FULL command now provides the value of the stored semantics tag as part of the file information returned to the user. This is the recommended method for quickly determining whether or not a file is tagged. The following display illustrates how the DIRECTORY/FULL command returns the RMS attributes for a DDIF file named X.DDIF.

```
X.DDIF;1                        File ID:  (767,20658,0)
.
.
.
RMS attributes:     Stored semantics: DDIF
.
.
.
```

A.1.1.2 ANALYZE/RMS_FILE

When you use the ANALYZE/RMS_FILE command to analyze a DDIF file, the utility returns the file tag as an RMS file attribute.

```
FILE HEADER
File Spec: USERD$:[TEST]X.DDIF;1
   .
   .
   .
Stored semantics: DDIF
   .
   .
   .
```

One ANALYZE/RMS_FILE command option is to create an output FDL file that reflects the results of the analysis.

```
$ ANALYZE/RMS_FILE/FDL filespec
```

When you use this option for analyzing a tagged file, the output FDL file includes the file tag as a secondary attribute to the FILE primary attribute. This is illlustrated in the following FDL file excerpt:

```
IDENT " 9-JUN-1988 13:27:30 VMS/VMS ANALYZE/RMS_FILE Utility"
   .
   .
   .
SYSTEM
        SOURCE                  VMS
FILE
        ALLOCATION              3
   .
   .
   .
        STORED_SEMANTICS        %X'2B0C8773010301' ! DDIF
   .
   .
   .
```

A.1.2 Creating RMS File Tags

The CDA$CREATE_FILE routine in the Compound Document Architecture toolkit creates and tags DDIF files. However, you may encounter a DDIF file that was created without a file tag or a DDIF file whose file tag was not preserved during file processing.

The DCL command SET FILE provides a new qualifier, /[NO]SEMANTICS, that permits you to tag a DDIF file through the DCL interface for VMS DECwindows systems. You can also use the qualifier to change a tag or to remove a tag from a file.

The following command line tags the file X.DDIF as a DDIF file by assigning the appropriate value to the /SEMANTICS qualifier:

```
$ SET FILE X.DDIF/SEMANTICS=DDIF
```

See Section A.1 for information about how to use logical name tables to assign a mnemonic to a tag.

A subsequent DIRECTORY/FULL command displays the following line as part of the file header:

```
.
.
.
RMS attributes:     Stored semantics: DDIF
.
.
.
```

The next example illustrates how to use the SET FILE command to delete an RMS file tag:

```
$ SET FILE X.DDIF/NOSEMANTICS
```

A.1.3 Preserving RMS File Tags and DDIF Semantics

The COPY command and the VMS Mail Utility preserve RMS file tags and DDIF semantics when you copy or mail a DDIF file on a VMS DECwindows system, except for conditions described in Sections A.1.3.1 and A.1.3.2.

The Backup Utility always preserves file tags and semantics when you back up a DDIF file to magnetic tape.

A.1.3.1 COPY Command

This section describes the results of using the COPY command with DDIF files for various operations.

When you copy a DDIF file to a disk on a VMS DECwindows system using the COPY command, VMS RMS preserves the DDIF tag and the DDIF semantics of the input file in the output file.

When you copy a DDIF file to a nondisk device on a VMS DECwindows system using the COPY command, VMS RMS does *not* preserve the DDIF tag or the DDIF semantics of the input file in the output file. Instead, VMS RMS writes the text from the input file to the output file as variable-length records.

When you copy two or more DDIF and text files in any combination to a single output file, the output file takes the characteristics of the first input file, as shown in the following examples.

1 In the first example, the first input file is a text file, so the output file (FOO.TXT) contains variable-length text records from X.TXT, Y.DDIF, and Z.TXT, but does not include the DDIF tag from Y.DDIF.

```
$ COPY X.TXT,Y.DDIF,Z.TXT FOO.TXT
```

2 In the next example, the first input file (A.DDIF) is a DDIF file, so the output file (FOO.DDIF) includes the DDIF tag as well as the DDIF semantics from A.DDIF. The attempt to copy the text input file (Z.TXT) fails because there is no Text-to-DDIF RMS extension, but the contents of B.DDIF and C.DDIF are copied to the output file. However, the output file has no practical use because, as a result of the way DDIF files are structured, only the data from the first input file (A.DDIF) is accessible in the output file.

```
$ COPY A.DDIF,B.DDIF,Z.TXT,C.DDIF FOO.DDIF
```

3 In the final example, the first input file (A.DDIF) is a DDIF file, so the output file (FOO.DDIF) includes the DDIF tag as well as the contents of A.DDIF. FOO.DDIF also includes the contents of B.DDIF and C.DDIF. Again, however, the output file has no practical use because, as a result of the way DDIF files are structured, only the data from the first input file (A.DDIF) is accessible in the output file.

```
$ COPY A.DDIF,B.DDIF,C.DDIF FOO.DDIF
```

A.1.3.2 VMS Mail Utility

The VMS Mail Utility preserves the DDIF file tag when DDIF files are mailed between systems running VMS DECwindows. The VMS Mail Utility also preserves the DDIF file tag when you create an output file on a VMS DECwindows system using the EXTRACT command.

When you read a mail message that is a DDIF file, the VMS Mail Utility outputs only the text portion of the file. Similarly, if you edit a DDIF mail file, you can access only the file text; the output file is a text file that can no longer be used as a DDIF file. However, if you forward a message that consists of a DDIF file, the VMS Mail Utility sends the entire DDIF file, including DDIF semantics and the DDIF tag, to the addressee.

A.1.4 APPEND Command

This section describes what happens when you attempt to use the APPEND command in conjunction with DDIF and text files.

In the first example, the APPEND command appends a DDIF file to a text file:

```
$ APPEND X.DDIF Y.TXT
```

The output file, Y.TXT, contains its original text records as well as text from the input file, X.DDIF, reformatted as variable-length text records.

In the next example, the APPEND command appends a DDIF file to another DDIF file:

```
$ APPEND X.DDIF Y.DDIF
```

The output file, Y.DDIF, contains the DDIF tag, the original contents of Y.DDIF, and the contents of X.DDIF. However, the portion of the file that contains X.DDIF is not accessible because of the way DDIF files are structured.

In the final example, the APPEND command attempts to append a text file to a DDIF file:

```
$ APPEND X.TXT Y.DDIF
```

This append operation fails because there is no Text-to-DDIF RMS extension.

A.2 DDIF Support in a Heterogeneous Environment

This section describes the implementation of DDIF support in two heterogeneous environments. The first heterogeneous environment includes VMS DECwindows systems and non-VMS systems. The second heterogeneous environment includes VMS DECwindows systems and VMS systems that do not support VMS DECwindows.

A.2.1 EXCHANGE/NETWORK Command

A new DCL command, EXCHANGE/NETWORK, has been created to support the transfer of files between VMS systems and non-VMS systems that do not support VMS file types. The EXCHANGE/NETWORK command transfers files in either record mode or block mode but can only be used when both systems support DECnet file transfers.

To interactively tag a DDIF file and transfer the file between a non-VMS operating system and a VMS system running DECwindows, do the following:

1 Create the following file, assigning it the name DDIF.FDL:

```
FILE
        ORGANIZATION            sequential
        STORED_SEMANTICS        DDIF

RECORD
        CARRIAGE_CONTROL        none
        FORMAT                  fixed
        SIZE                    512
```

2 Use the following DCL command to transfer the desired file:

```
$ EXCHANGE/NETWORK/FDL=DDIF.FDL input_filespec output_filespec
```

A.2.2 Using the COPY Command in a Heterogeneous Environment

If you use the COPY command to copy tagged DDIF files to systems other than VMS DECwindows systems, the results will vary depending on the target system:

- If the target system is a non-VMS system, the file is copied, but the DDIF tag is not preserved.
- If the target system is a VMS system that does not support VMS DECwindows, the copy operation fails.

A.2.3 VMS Mail Utility in a Heterogeneous Environment

If you try to send mail messages containing DDIF files to non-VMS systems that do not support tagged files, the VMS Mail Utility returns the NOACCEPTMSG error message, indicating that the remote node cannot accept the message format.

Similarly, the VMS Mail Utility does not support the mailing of DDIF files to VMS systems that do not support VMS DECwindows. As with non-VMS systems, the VMS Mail Utility returns the NOACCEPTMSG error message, indicating that the remote node cannot accept the message format.

A.3 VMS RMS Interface Changes

This section provides details about the changes made to the VMS RMS interface that support access to text in VMS DECwindows DDIF files. It includes information related to tagging files and accessing tagged files through the VMS RMS interface. The section also describes how tags are preserved at the VMS RMS interface.

A.3.1 Programming Interface for File Tagging

This appendix focuses on the use of the DDIF tag for supporting VMS DECwindows files, although VMS RMS also supports file tagging for other compound document data formats.

You can tag a file from the VMS RMS interface by using the $CREATE service in conjunction with a new extended attribute block (XAB) called the item XAB ($XABITM). The $XABITM macro is a general-purpose macro that was added to the RMS interface to support several Version 5.0 features. Tagged file support involves the use of the two item codes shown in Table A–1.

Table A–1 Tag Support Item Codes

Item	Buffer Size	Function
XAB$_STORED_SEMANTICS	64 bytes maximum	Defines the file semantics established when the file is created
XAB$_ACCESS_SEMANTICS	64 bytes maximum	Defines the file semantics desired by the accessing program

The entries XAB$_STORED_SEMANTICS and XAB$_ACCESS_SEMANTICS in the item list can represent either a control (set) function or a monitor (sense) function that can be passed to VMS RMS from the application program by way of the RMS interface.

The symbolic value XAB$K_SEMANTICS_MAX_LEN represents the tag length. This value may be used to allocate buffer space for sensing and setting stored semantics for the DDIF file.

Within any one $XABITM, you can activate either the set function or the sense function for the XAB$_STORED_SEMANTICS and XAB$_ACCESS_SEMANTICS items, because a common field (XAB$B_MODE) determines which function is active. If you want to activate both the set function and the sense function for either or both items, you must use two $XABITM control blocks, one for setting the functions and one for sensing the functions.

Each entry in the item list addressed by the $XABITM is made up of three longwords and a longword 0 terminates the list. You can locate the item list anywhere within the readable address space for a process, but any buffers required by the related function must be located in read/write memory. If the item list is invalid, RMS returns a status of RMS$_XAB in the RAB$L_STS field and the address of the XAB in RAB$L_STV.

The format and arguments of the $XABITM macro are as follows. Note that the block length field and the type code field are statically initialized by the $XABITM macro, or may be explicitly initialized using a high-level language.

FORMAT

$XABITM *ITEMLIST=item-list-address,*
MODE= { *sensemode* / *setmode* }*,*
NXT=next-xab-address

Arguments

The ITEMLIST argument defaults to 0 but a valid pointer must be specified when you use a XABITM. MODE defaults to *sensemode*. The symbolic offset, size, and a brief description of each XABITM field are described in the following list:

- The block length field (XAB$B_BLN) is a 1-byte static field that defines the length of the XABITM, in bytes. This field is initialized to the value XAB$C_ITMLEN.
- The type code (XAB$B_COD) field is a 1-byte static field that identifies this control block as a XABITM. This field is initialized to the value XAB$C_ITM.
- The XAB$L_ITEMLIST field is a longword field that contains the symbolic address of the item list.
- The XAB$B_MODE field is a 1-byte field that specifies whether or not the items can be set by the program. It contains either the symbolic value XAB$K_SETMODE or the symbolic value XAB$K_SENSEMODE (default).
- The XAB$L_NXT field is a longword field that contains the symbolic address of the next XAB in the XAB chain. A value of 0 (the default) indicates that the current XAB is the last (or only) XAB in the chain.

Example A–1 illustrates a BLISS–32 program that tags a file through the RMS interface. The tag value shown is a 6-byte hexadecimal number representing the code for the DDIF tag. The VMS RMS program interface accepts only hexadecimal tag values.

To write to a tagged file without using an RMS extension, the application program must specify access semantics that match the file's stored semantics. As shown in the example, the $CREATE service tags the file and the $CONNECT service specifies the appropriate access semantics.

Example A–1 Tagging a File

```
MODULE TYPE$MAIN (
        IDENT = 'X-1',
        MAIN = MAIN,
        ADDRESSING_MODE (EXTERNAL=GENERAL)
        ) =
BEGIN
!
FORWARD ROUTINE
    MAIN : NOVALUE;                          ! Main routine
!
! INCLUDE FILES:
!
LIBRARY 'SYS$LIBRARY:LIB';
OWN
    NAM             : $NAM(),
    RETLEN,
    DDIF_TAG         : BLOCK[ 7, BYTE]
                INITIAL( BYTE( %X'2B', %X'0C', %X'87' %X'73', %X'01',
                               %X'03', %X'01')),
    FAB_XABITM         :
                $xabitm
                  ( itemlist=
                        $ITMLST_UPLIT
                          (
                            (ITMCOD=XAB$_STORED_SEMANTICS,
                             BUFADR=DDIF_TAG,
                             BUFSIZ=%ALLOCATION(DDIF_TAG))
                          ),
                     mode = SETMODE),
    RAB_XABITM         :
                $xabitm
                  ( itemlist=
                        $ITMLST_UPLIT
                          (
                            (ITMCOD=XAB$_ACCESS_SEMANTICS,
                             BUFADR=DDIF_TAG,
                             BUFSIZ=%ALLOCATION(DDIF_TAG))
                          ),
                     mode = SETMODE),
    FAB          : $FAB( fnm = 'TAGGED-FILE.TEST',
                         nam = NAM,
                         mrs = 512,
                         rfm = FIX,
                         fac = <GET,PUT,UPD>,
                         xab = FAB_XABITM),
    REC              : BLOCK[512,BYTE],
    STATUS,
```

(continued on next page)

Example A–1 (Cont.) Tagging a File

```
    RAB         : $RAB( xab = RAB_XABITM,
                        fab = FAB,
                        rsz = 512,
                        rbf = REC,
                        usz = 512,
                        ubf = REC),
    DESC            : BLOCK[8,BYTE] INITIAL(0);
ROUTINE MAIN : NOVALUE =
BEGIN
STATUS = $CREATE( FAB = FAB );
IF NOT .STATUS
THEN
    SIGNAL (.STATUS);
STATUS = $CONNECT( RAB = RAB );
IF NOT .STATUS
THEN
    SIGNAL (.STATUS);
STATUS = $CLOSE( FAB = FAB );
IF NOT .STATUS
THEN
    SIGNAL (.STATUS);
END;
END
ELUDOM
```

A.3.2 Accessing a Tagged File

This section provides details of how VMS RMS handles access to tagged files at the program level. When a program accesses a tagged file, VMS RMS must determine whether and when to associate an RMS extension with the access. This is important to the programmer because an RMS extension may change the attributes of the accessed file.

For example, a DDIF file is stored as a sequentially organized file having 512-byte, fixed-length records. If the DDIF-to-Text RMS extension is used to extract text data from a DDIF file, the accessed file appears as a sequentially organized file having variable-length records with an implicit carriage return.

One consideration in determining whether an access requires the RMS extension is the type of access (FAB$B_FAC). When an application program opens a file through the VMS RMS program interface, it must specify whether it will be doing record I/O (default), block I/O (BIO), or mixed I/O (BRO), where the program has the option of using either block I/O or record I/O for each access. For example, if block I/O operations are specified, VMS RMS does not associate the RMS extension with the file access.

Another consideration is whether the program senses the tag when it opens a file. If the program does not sense the tag when it opens a DDIF file for record access, VMS RMS associates the RMS extension during the $OPEN and returns the file attributes that have been modified by the extension.

The final consideration is the access semantics the program specifies and the file's stored semantics (tag). If the program specifies block I/O (FAB$V_BIO) operations, RMS does not associate the RMS extension and the $OPEN service returns the file's stored attributes to the accessing program regardless of whether the program senses tags.

A.3.2.1 File Accesses That Do Not Sense Tags

This section describes what happens when a program does not use the XABITM to sense a tag when it opens a file.

When a program opens a DDIF file for record operations and does not sense the tag, VMS RMS assumes that the program wants to access text data in the file. In this case, VMS RMS associates the RMS extension, which provides file attributes that correspond to record-mode access.

When a program opens a DDIF file with the FAB$V_BRO option and does not sense the tag, any subsequent attempt to use block I/O fails. If the program specifies block I/O (FAB$V_BIO) when it invokes the $CONNECT service, the operation fails because the file attributes returned at $OPEN permit record access only. Similarly, if the program specifies the FAB$V_BRO option when it opens the file, and then specifies mixed mode (block/record) operations by not specifying RAB$V_BIO at $CONNECT time, block operations such as READ and WRITE are disallowed.

A.3.2.2 File Accesses That Sense Tags

VMS RMS does not associate the RMS extension as part of the $OPEN service if a program opens a DDIF file and senses the stored semantics. This allows the program to specify access semantics with the $CONNECT service. VMS RMS returns the file attributes, including the stored semantics attribute (tag value), to the program as part of the $OPEN service.

When the program subsequently invokes the $CONNECT service, VMS RMS uses the specified operations mode to determine its response. If the program specified FAB$V_BRO with the $OPEN service and then specifies block I/O (RAB$V_BIO) when it invokes the $CONNECT service, VMS RMS does not associate the RMS extension.

But if the program specifies record access or FAB$V_BRO when it opens the file and then decides to use record I/O when it invokes the $CONNECT service, VMS RMS compares the access semantics with the file's stored semantics to determine whether to associate the RMS extension. If the access semantics match the stored semantics, VMS RMS does not associate the RMS extension. If the access semantics do not match the stored semantics, VMS RMS associates the access with the RMS extension. In this case, the program must use the $DISPLAY service to obtain the modified file attributes. If VMS RMS cannot find the appropriate RMS extension, the operation fails and the $CONNECT service returns the EXTNOTFOU error message.

If the application program senses the file's stored semantics, VMS RMS allows mixed-mode operations. In this case, mixed block and record operations are permitted because the application gets record mode file attributes and data from the RMS extension and block mode file attributes and data from the file.

Example A–2 illustrates a BLISS–32 program that accesses a tagged file from an application program that does not use an RMS extension.

Example A–2 Accessing a Tagged File

```
MODULE TYPE$MAIN (
        IDENT = 'X-1',
        MAIN = MAIN,
        ADDRESSING_MODE (EXTERNAL=GENERAL)
        ) =
BEGIN
!
FORWARD ROUTINE
    MAIN : NOVALUE;                          ! Main routine
!
! INCLUDE FILES:
!
LIBRARY 'SYS$LIBRARY:STARLET';
OWN
    NAM             : $NAM(),
    ITEM_BUFF    : BLOCK[ XAB$K_SEMANTICS_MAX_LEN,BYTE ],
    RETLEN,
    FAB_XABITM         :
                $xabitm
                  ( itemlist=
                        $ITMLST_UPLIT
                          ((ITMCOD=XAB$_STORED_SEMANTICS,
                             BUFADR=ITEM_BUFF,
                             BUFSIZ=XAB$K_SEMANTICS_MAX_LEN,
                             RETLEN=RETLEN)),
                    mode = SENSEMODE),
    RAB_ITEMLIST  : BLOCK[ ITM$S_ITEM + 4, BYTE ],
    RAB_XABITM    : $XABITM
                  ( itemlist=RAB_ITEMLIST,
                    mode=SETMODE ),
    FAB           : $FAB( fnm = 'TAGGED-FILE.TEST',
                        nam = NAM,
                        fac = <GET,PUT,UPD>,
                        xab = FAB_XABITM),
    REC             : BLOCK[512,BYTE],
    STATUS,
    RAB           : $RAB( xab = RAB_XABITM,
                        fab = FAB,
                        rsz = 512,
                        rbf = REC,
                        usz = 512,
                        ubf = REC),
    DESC            : BLOCK[8,BYTE] INITIAL(0);
ROUTINE MAIN : NOVALUE =
BEGIN
STATUS = $OPEN( FAB = FAB );
IF NOT .STATUS
THEN
    SIGNAL (.STATUS);
RAB_ITEMLIST[ ITM$W_BUFSIZ ] = .RETLEN;
RAB_ITEMLIST[ ITM$L_BUFADR ] = ITEM_BUFF;
RAB_ITEMLIST[ ITM$W_ITMCOD ] = XAB$_ACCESS_SEMANTICS;
STATUS = $CONNECT( RAB = RAB );
IF NOT .STATUS
THEN
```

(continued on next page)

Example A–2 (Cont.) Accessing a Tagged File

```
    SIGNAL (.STATUS);
STATUS = $CLOSE( FAB = FAB );
IF NOT .STATUS
THEN
    SIGNAL (.STATUS);
END;
END
ELUDOM
```

A.3.3 Preserving Tags

In order to preserve the integrity of a tagged file that is being copied or transmitted, the tag must be preserved in the destination (output) file. The most efficient way to use the RMS interface for propagating tags is to open the source file (input) and sense the tag using a $XABITM with the item code XAB$_STORED_SEMANTICS:

```
    .
    .
    .
ITEMLIST[ ITM$W_BUFSIZ ] = XAB$K_SEMANTICS_MAX_LEN;
ITEMLIST[ ITM$L_BUFADR ] = ITEM_BUFF;
ITEMLIST[ ITM$L_RETLEN ] = RETLEN;
ITEMLIST[ ITM$W_ITMCOD ] = XAB$_STORED_SEMANTICS;
    .
    .
    .
XABITM[ XAB$B_MODE ] = XAB$K_SENSEMODE;
STATUS = $OPEN( FAB = FAB );
    .
    .
    .
```

Then create the destination (output) file and set the tag using a $XABITM with the item code XAB$_STORED_SEMANTICS:

```
    .
    .
    .
IF .RETLEN GTR 0
THEN
    BEGIN
    ITEMLIST[ ITM$W_ITMCOD ] = XAB$_STORED_SEMANTICS;
    ITEMLIST[ ITM$L_SIZE   ] = .RETLEN;
    XABITM[ XAB$B_MODE ] = XAB$K_SETMODE;
    END;

STATUS = $CREATE( FAB = FAB );
    .
    .
    .
END;
END
ELUDOM
```

A.4 Distributed File System Support for DDIF Tagged Files

Version 1.1 of the Distributed File System (DFS) includes limited support for DDIF tagged files. You can create and read DDIF files on a DFS device when the DFS client node is running VMS DECwindows. You can also use the DIRECTORY/FULL command to determine whether or not a DDIF file on a DFS device is tagged.

You cannot use the SET FILE/[NO]SEMANTICS command either to tag DDIF files or to remove the tags from DDIF files on a DFS device. Furthermore, the Backup Utility does not preserve the DDIF tag or the DDIF stored semantics for data files on a DFS device.

A.5 VMS RMS Errors

Four VMS RMS error messages signal the user when the appropriate error condition exists:

- RMS$_EXTNOTFOU
- RMS$_SEMANTICS
- RMS$_EXT_ERR
- RMS$_OPNOTSUP

The RMS$_EXTNOTFOU error message indicates that VMS RMS has not found the specified RMS extension. Verify that the file is correctly tagged, using the DIRECTORY/FULL command, and that the application program is specifying the appropriate access semantics.

VMS RMS returns the RMS$_SEMANTICS error message when you try to create a tagged file on a remote VMS system that does not support VMS DECwindows from a system that does support VMS DECwindows.

VMS RMS returns the RMS$_EXT_ERR error when the DDIF RMS extension detects an inconsistency.

VMS RMS returns the RMS$_OPNOTSUP error when the RMS DDIF extension is invoked by an RMS operation. For example, if the extension does not support write access to a DDIF file, verify that the application program is not performing record operations that modify the file.

B CDA Toolkit Example Program

This appendix illustrates a sample program, written in VAX C, that uses the CDA Toolkit to create a DDIF document. Example B–1 contains comments where necessary, and Example B–2 illustrates the analysis output of the DDIF document created by the program. The callouts in this example correspond to the callouts in Example B–2. For example, if a callout corresponds to a call to the CREATE ROOT AGGREGATE routine in Example B–1, the callout in Example B–2 identifies the beginning of the document root aggregate created by that call.

Example B–1 Sample CDA Toolkit Program

```
#include <ddif$def.h>          /* Include DDIF keyword definitions.        */
#include <cda$def.h>           /* Include CDA Toolkit keyword definitions. */

#define FAILURE(x)     (((x) & 1) == 0)

/*
** Subroutines to generate frequently-used aggregates.
*/

extern unsigned long create_sft_dir( );
extern unsigned long create_hrd_dir( );
extern unsigned long create_gtx( );

main()
{
unsigned long   aggregate_type;
unsigned long   aggregate_item;
unsigned long   aggregate_index;
unsigned long   add_info;
unsigned long   buffer_length;
unsigned long   file_handle;
unsigned long   integer_value;
unsigned long   integer_length = sizeof( integer_value );
unsigned long   local_length;
unsigned long   status;
unsigned long   stream_handle;

unsigned long   aggregate_handle;
unsigned long   aggregate_handle_length = sizeof( aggregate_handle );
unsigned long   root_aggregate_handle;
unsigned long   previous_aggregate_handle;

unsigned long   aggregate_handle_stack[ 100 ];
unsigned long   ahs_index = 0;
unsigned long   document_type;

/* Data and structures for the frame definition. */

struct frm_flags        sga_frame_flags;

unsigned long   frame_ur_x_value = 6000;
unsigned long   frame_ur_y_value = 2400;
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
/* Data for the polyline and Bezier curve. */

#define MAX_POINTS 4
unsigned long   i;
unsigned long   poly_points[MAX_POINTS][2] =
                        {
                        { 500, 500 },
                        { 2500, 2000 },
                        { 3500, 2000 },
                        { 5500, 500 }
                        };

/* Data for the arc. */

struct arc_flags set_arc_flags;

float   arc_start = 4.5e1;
float   arc_extent = 9.0e1;

unsigned long   arc_line_width = 60;

unsigned char   filename[] = "DDIF_EXAMPLE.DDIF";
unsigned long   filename_length = sizeof( filename ) - 1;
unsigned char   result_file_spec[255];
unsigned long   result_file_spec_len = sizeof( result_file_spec );
unsigned long   result_file_ret_len;

unsigned long   dsc_major_version = 1;
unsigned long   dsc_major_version_length = sizeof( dsc_major_version );

unsigned long   dsc_minor_version = 0;
unsigned long   dsc_minor_version_length = sizeof( dsc_minor_version );

unsigned char   dsc_product_identifier[] = "DDIF$";
unsigned long   dsc_product_identifier_length =
                        sizeof( dsc_product_identifier ) - 1;

unsigned char   dsc_product_name[] = "Test V1.0";
unsigned long   dsc_product_name_length = sizeof( dsc_product_name ) - 1;

unsigned char   dhd_languages_1[] = "E/USA/";
unsigned long   dhd_languages_length_1 = sizeof( dhd_languages_1 ) - 1;

unsigned char   dhd_languages_2[] = "Mandarin";
unsigned long   dhd_languages_length_2 = sizeof( dhd_languages_2 ) - 1;

unsigned char   sga_content_category_1[] = "$T";
unsigned long   sga_content_category_length_1 =
                        sizeof( sga_content_category_1 ) - 1;

unsigned char   sga_content_category_2[] = "$2D";
unsigned long   sga_content_category_length_2 =
                        sizeof( sga_content_category_2 ) - 1;

unsigned char   txt_content[] = "This is the first line of the example text.";
unsigned long   txt_content_length = sizeof( txt_content ) - 1;

unsigned char   gtx_content_1[] = "This is the second line of the example text,
                and should be separated from the first line by a single space.";
unsigned long   gtx_content_length_1 = sizeof( gtx_content_1 ) - 1;

unsigned char   gtx_content_2[] = "The third line of the example text will
                begin on a new line.";
unsigned long   gtx_content_length_2 = sizeof( gtx_content_2 ) - 1;
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
unsigned char   gtx_content_3[] = "The fourth line of the example text will be
                separated from the previous lines by a blank line.";
unsigned long   gtx_content_length_3 = sizeof( gtx_content_3 ) - 1;

unsigned char   tyd_label[] = "FRAME";
unsigned long   tyd_label_length =
                        sizeof( tyd_label ) - 1;

unsigned char   pline_label[] = "pline";
unsigned long   pline_label_length =
                        sizeof( pline_label ) - 1;

unsigned char   bline_label[] = "bline";
unsigned long   bline_label_length =
                        sizeof( bline_label ) - 1;

unsigned char   filled_arc_label[] = "filled_arc";
unsigned long   filled_arc_label_length =
                        sizeof( filled_arc_label ) - 1;

    printf("Creating in-memory DDIF structure...\n" );


/*
**************************************************************************
**
**  The overall structure is as follows:
**
**                      Root
**                   Aggregate
**                      / | \
**                     /  |  \
**                    /   |   \
**                   /    |    \
**                  /     |     \
**           Descriptor  Header  Content
**
**************************************************************************
*/

    /*
    ** Create the DDIF Root Aggregate.
    */

    aggregate_type = DDIF$_DDF;
    status = cda$create_root_aggregate( 0, 0, 0, 0,                    ❶
                                        &aggregate_type,
                                        &root_aggregate_handle );
    if( FAILURE( status ) )
        return ( status );
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
/*
*****************************************************************************
**
**  DESCRIPTOR:
**
**      1) create the Descriptor aggregate
**      2) attach it to the Root aggregate
**      3) fill in the items in the Descriptor aggregate.
**
*****************************************************************************
*/

    /*
    ** Create the Descriptor aggregate and attach it to the Root aggregate
    ** by storing its handle in the Descriptor item of the Root aggregate.
    */

    aggregate_type = DDIF$_DSC;
    status = cda$create_aggregate( &root_aggregate_handle,          ❷
                                   &aggregate_type,
                                   &aggregate_handle_stack[ahs_index] );
    if( FAILURE( status ) )
        return ( status );

    aggregate_item = DDIF$_DDF_DESCRIPTOR;
    status = cda$store_item( &root_aggregate_handle,                ❸
                             &root_aggregate_handle,
                             &aggregate_item,
                             &aggregate_handle_length,
                             &aggregate_handle_stack[ahs_index] );
    if( FAILURE( status ) )
        return ( status );

    /*
    ** Fill in the Major Version item of the Descriptor aggregate.
    */

    aggregate_item = DDIF$_DSC_MAJOR_VERSION;
    status = cda$store_item( &root_aggregate_handle,                ❹
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &dsc_major_version_length,
                             &dsc_major_version );
    if( FAILURE( status ) )
        return ( status );

    /*
    ** Fill in the Minor Version item of the Descriptor aggregate.
    */

    aggregate_item = DDIF$_DSC_MINOR_VERSION;
    status = cda$store_item( &root_aggregate_handle,
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &dsc_minor_version_length,
                             &dsc_minor_version );
    if( FAILURE( status ) )
        return ( status );
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
    /*
    ** Fill in the Product Identifier item of the Descriptor aggregate.
    */
    aggregate_item = DDIF$_DSC_PRODUCT_IDENTIFIER;
    status = cda$store_item( &root_aggregate_handle,
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &dsc_product_identifier_length,
                              dsc_product_identifier );
    if( FAILURE( status ) )
        return ( status );

    /*
    ** Fill in the Product Name item of the Descriptor aggregate.
    */
    aggregate_index = 0;
    aggregate_item = DDIF$_DSC_PRODUCT_NAME;
    add_info = CDA$K_ISO_LATIN1;
    status = cda$store_item( &root_aggregate_handle,                    ❺
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &dsc_product_name_length,
                              dsc_product_name,
                             &aggregate_index,
                             &add_info );
    if( FAILURE( status ) )
        return ( status );

/*
*****************************************************************************
**
**  HEADER:
**
**      1) create the Header aggregate
**      2) attach it to the Root aggregate
**      3) fill in the items in the Header aggregate
**
*****************************************************************************
*/

    /*
    ** Create the Header aggregate and attach it to the Root aggregate
    ** by storing its handle in the Header item of the Root aggregate.
    */
    aggregate_type = DDIF$_DHD;
    status = cda$create_aggregate( &root_aggregate_handle,          ❻
                                   &aggregate_type,
                                   &aggregate_handle_stack[ahs_index] );
    if( FAILURE( status ) )
        return ( status );
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
aggregate_item = DDIF$_DDF_HEADER;
status = cda$store_item( &root_aggregate_handle,                    ❼
                         &root_aggregate_handle,
                         &aggregate_item,
                         &aggregate_handle_length,
                         &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/*
** Fill in the Languages item in Header aggregate.  First, the enumeration
** value must be stored, then the data value.  An index must be used since
** these are arrays.
*/

aggregate_item = DDIF$_DHD_LANGUAGES_C;
integer_value = DDIF$K_ISO_639_LANGUAGE;
aggregate_index = 0;
status = cda$store_item( &root_aggregate_handle,                    ❽
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value,
                         &aggregate_index );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_DHD_LANGUAGES;
local_length = 7;
aggregate_index = 0;
status = cda$store_item( &root_aggregate_handle,                    ❾
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &dhd_languages_length_1,
                          dhd_languages_1,
                         &aggregate_index );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_DHD_LANGUAGES_C;
integer_value = DDIF$K_OTHER_LANGUAGE;
aggregate_index = 1;
status = cda$store_item( &root_aggregate_handle,                    ❿
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value,
                         &aggregate_index );
if( FAILURE( status ) )
    return ( status );
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
    aggregate_item = DDIF$_DHD_LANGUAGES;
    integer_value = DDIF$K_OTHER_LANGUAGE;
    aggregate_index = 1;
    add_info = CDA$K_ISO_LATIN1;
    status = cda$store_item( &root_aggregate_handle,                    (11)
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &dhd_languages_length_2,
                              dhd_languages_2,
                             &aggregate_index,
                             &add_info );
    if( FAILURE( status ) )
        return ( status );

/*
**************************************************************************
**
**  CONTENT:
**
**      1) create the Segment aggregate
**      2) attach it to the Root aggregate
**      3) fill in the items in the Segment aggregate
**
**************************************************************************
*/

    /*
    ** Create the Segment aggregate and attach it to the Root aggregate
    ** by storing its handle in the Content item of the Root aggregate.
    */

    aggregate_type = DDIF$_SEG;
    status = cda$create_aggregate( &root_aggregate_handle,              (12)
                                   &aggregate_type,
                                   &aggregate_handle_stack[ahs_index] );
    if( FAILURE( status ) )
        return ( status );

    aggregate_item = DDIF$_DDF_CONTENT;
    status = cda$store_item( &root_aggregate_handle,                    (13)
                             &root_aggregate_handle,
                             &aggregate_item,
                             &aggregate_handle_length,
                             &aggregate_handle_stack[ahs_index] );
    if( FAILURE( status ) )
        return ( status );

    /*
    ** Now fill in the items in the Segment aggregate.
    */

    ahs_index++;
    aggregate_type = DDIF$_SGA;
    status = cda$create_aggregate( &root_aggregate_handle,              (14)
                                   &aggregate_type,
                                   &aggregate_handle_stack[ahs_index] );
    if( FAILURE( status ) )
        return ( status );
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
aggregate_item = DDIF$_SEG_SPECIFIC_ATTRIBUTES;
status = cda$store_item( &root_aggregate_handle,                    ⓯
                         &aggregate_handle_stack[ahs_index-1],
                         &aggregate_item,
                         &aggregate_handle_length,
                         &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/*
** Store content category in specific-attribute aggregate.
*/

aggregate_item = DDIF$_SGA_CONTENT_CATEGORY;
aggregate_index = 0;
add_info = DDIF$K_T_CATEGORY;
status = cda$store_item( &root_aggregate_handle,                    ⓰
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &sga_content_category_length_1,
                          sga_content_category_1,
                         &aggregate_index,
                         &add_info );
if( FAILURE( status ) )
    return ( status );

ahs_index--;       /* End of segment attributes section */

/*
** Create Text Content aggregate and store its handle in the SEG_CONTENT
** item in DDF_CONTENT.  (This is the first aggregate in a Sequence Of,
** so it is attached with a store. The rest will be inserted.)
*/

ahs_index++;
aggregate_type = DDIF$_TXT;
status = cda$create_aggregate( &root_aggregate_handle,              ⓱
                               &aggregate_type,
                               &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_SEG_CONTENT;
status = cda$store_item( &root_aggregate_handle,                    ⓲
                         &aggregate_handle_stack[ahs_index-1],
                         &aggregate_item,
                         &aggregate_handle_length,
                         &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/*
** Add a text line.
*/
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
aggregate_item = DDIF$_TXT_CONTENT;
status = cda$store_item( &root_aggregate_handle,                    (19)
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &txt_content_length,
                          txt_content );
if( FAILURE( status ) )
    return ( status );

/* Save the handle of the segment_content aggregate. */

previous_aggregate_handle = aggregate_handle_stack[ahs_index];

/* Insert a space (hard) directive. */

status = create_hrd_dir ( &root_aggregate_handle,                   (20)
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_SPACE );
if( FAILURE( status ) )
    return ( status );

/* Create a General Text Content aggregate. */

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_gtx ( &root_aggregate_handle,                       (21)
                      &previous_aggregate_handle,
                      &aggregate_handle_stack[ahs_index],
                       gtx_content_1 );
if( FAILURE( status ) )
    return ( status );

/* Insert a new-line (soft) directive to force a new line. */

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_sft_dir ( &root_aggregate_handle,                   (22)
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
if( FAILURE( status ) )
    return ( status );

/* Create another General Text Content aggregate. */

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_gtx ( &root_aggregate_handle,                       (23)
                      &previous_aggregate_handle,
                      &aggregate_handle_stack[ahs_index],
                       gtx_content_2 );
if( FAILURE( status ) )
    return ( status );

/* Insert two new-line directives to cause a skipped line. */
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_sft_dir ( &root_aggregate_handle,                    (24)
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
if( FAILURE( status ) )
    return ( status );

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_sft_dir ( &root_aggregate_handle,
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
if( FAILURE( status ) )
    return ( status );

/* Create another General Text Content aggregate. */

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_gtx ( &root_aggregate_handle,                        (25)
                      &previous_aggregate_handle,
                      &aggregate_handle_stack[ahs_index],
                       gtx_content_3 );
if( FAILURE( status ) )
    return ( status );

/* Insert two new-line directives to cause a skipped line. */

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_sft_dir ( &root_aggregate_handle,                    (26)
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
if( FAILURE( status ) )
    return ( status );

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_sft_dir ( &root_aggregate_handle,
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
if( FAILURE( status ) )
    return ( status );

/* Insert next general-text line. */

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_gtx ( &root_aggregate_handle,                        (27)
                      &previous_aggregate_handle,
                      &aggregate_handle_stack[ahs_index],
                       "The following is a polyline within a frame:" );
if( FAILURE( status ) )
    return ( status );

/* Insert two new-line directives to cause a skipped line. */
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_sft_dir ( &root_aggregate_handle,                    (28)
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
if( FAILURE( status ) )
    return ( status );

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_sft_dir ( &root_aggregate_handle,
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
if( FAILURE( status ) )
    return ( status );

/*
** Create a new segment aggregate for the next part of the example.
*/

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
aggregate_type = DDIF$_SEG;
status = cda$create_aggregate( &root_aggregate_handle,               (29)
                               &aggregate_type,
                               &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/* Insert after previous aggregate */

status = cda$insert_aggregate( &aggregate_handle_stack[ahs_index],
                               &previous_aggregate_handle );
if( FAILURE( status ) )
    return ( status );

/*
** Create new segment attributes aggregate to define a galley frame.
** Store it in the segment aggregate just created.
*/

ahs_index++;
aggregate_type = DDIF$_SGA;
status = cda$create_aggregate( &root_aggregate_handle,               (30)
                               &aggregate_type,
                               &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_SEG_SPECIFIC_ATTRIBUTES;
status = cda$store_item( &root_aggregate_handle,                     (31)
                         &aggregate_handle_stack[ahs_index-1],
                         &aggregate_item,
                         &aggregate_handle_length,
                         &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
/*
** Create a type-definition aggregate and attach to the segment
** attribute aggregate.
*/

ahs_index++;
aggregate_type = DDIF$_TYD;
status = cda$create_aggregate( &root_aggregate_handle,          (32)
                               &aggregate_type,
                               &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_SGA_TYPE_DEFNS;
status = cda$store_item( &root_aggregate_handle,                (33)
                         &aggregate_handle_stack[ahs_index-1],
                         &aggregate_item,
                         &aggregate_handle_length,
                         &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/* Now store the type-definition label. */

aggregate_item = DDIF$_TYD_LABEL;
status = cda$store_item( &root_aggregate_handle,                (34)
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &tyd_label_length,
                          tyd_label );
if( FAILURE( status ) )
    return ( status );

/*
** Create an attribute aggregate, and attach to the
** type-def aggregate.
*/

ahs_index++;
aggregate_type = DDIF$_SGA;
status = cda$create_aggregate( &root_aggregate_handle,          (35)
                               &aggregate_type,
                               &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_TYD_ATTRIBUTES;
status = cda$store_item( &root_aggregate_handle,                (36)
                         &aggregate_handle_stack[ahs_index-1],
                         &aggregate_item,
                         &aggregate_handle_length,
                         &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/*
** Now that the type-def attributes aggregate is in place, store
** each desired attribute there.
*/
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
aggregate_item = DDIF$_SGA_CONTENT_CATEGORY;
aggregate_index = 0;
add_info = DDIF$K_2D_CATEGORY;
status = cda$store_item( &root_aggregate_handle,                    (87)
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &sga_content_category_length_2,
                          sga_content_category_2,
                         &aggregate_index,
                         &add_info );
if( FAILURE( status ) )
    return ( status );

/* Store the flags, indicating border on frame. */

aggregate_item = DDIF$_SGA_FRM_FLAGS;
sga_frame_flags.ddif$v_flow_around = 0;
sga_frame_flags.ddif$v_frame_border = 1;
sga_frame_flags.ddif$v_frame_background_fill = 0;
sga_frame_flags.ddif$v_frm_fill = 0;
integer_length = sizeof( sga_frame_flags );
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &sga_frame_flags );
if( FAILURE( status ) )
    return ( status );

/* Store the bounding coordinates of the frame. (Note indexing.) */

aggregate_item = DDIF$_SGA_FRM_BOX_LL_X_C;
integer_value = DDIF$K_VALUE_CONSTANT;
integer_length = sizeof( integer_value );
aggregate_index = 0;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value,
                         &aggregate_index );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_SGA_FRM_BOX_LL_X;
aggregate_index = 0;
integer_value = 0;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value,
                         &aggregate_index );
if( FAILURE( status ) )
    return ( status );
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
aggregate_item = DDIF$_SGA_FRM_BOX_LL_Y_C;
integer_value = DDIF$K_VALUE_CONSTANT;
aggregate_index = 1;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value,
                         &aggregate_index );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_SGA_FRM_BOX_LL_Y;
aggregate_index = 1;
integer_value = 0;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value,
                         &aggregate_index );
if( FAILURE( status ) )
    return ( status );

/* And now the upper-right coordinates... */

aggregate_item = DDIF$_SGA_FRM_BOX_UR_X_C;
integer_value = DDIF$K_VALUE_CONSTANT;
aggregate_index = 0;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value,
                         &aggregate_index );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_SGA_FRM_BOX_UR_X;
aggregate_index = 0;
integer_value = frame_ur_x_value;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value,
                         &aggregate_index );
if( FAILURE( status ) )
    return ( status );
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
aggregate_item = DDIF$_SGA_FRM_BOX_UR_Y_C;
integer_value = DDIF$K_VALUE_CONSTANT;
aggregate_index = 1;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value,
                         &aggregate_index );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_SGA_FRM_BOX_UR_Y;
aggregate_index = 1;
integer_value = frame_ur_y_value;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value,
                         &aggregate_index );
if( FAILURE( status ) )
    return ( status );

/* Now store the form-position item. */

aggregate_item = DDIF$_SGA_FRM_POSITION_C;
integer_value = DDIF$K_FRAME_GALLEY;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value );
if( FAILURE( status ) )
    return ( status );

ahs_index--; /* End of type attributes. */
ahs_index--; /* End of type-definition */
ahs_index--; /* End of segment attributes aggregate. */

/*
** Create a new segment aggregate in which to define the polyline,
** and store as the segment content.
*/
ahs_index++;
aggregate_type = DDIF$_SEG;
status = cda$create_aggregate( &root_aggregate_handle,          (38)
                               &aggregate_type,
                               &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/* Store into this segment. */
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
aggregate_item = DDIF$_SEG_CONTENT;
status = cda$store_item( &root_aggregate_handle,                    (39)
                         &aggregate_handle_stack[ahs_index-1],
                         &aggregate_item,
                         &aggregate_handle_length,
                         &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/* Store the segment ID. */

aggregate_item = DDIF$_SEG_ID;
status = cda$store_item( &root_aggregate_handle,                    (40)
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &pline_label_length,
                          pline_label );
if( FAILURE( status ) )
    return ( status );

/* Store the segment type ("FRAME"). */

aggregate_item = DDIF$_SEG_SEGMENT_TYPE;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &tyd_label_length,
                          tyd_label );
if( FAILURE( status ) )
    return ( status );

/* Create a Polyline aggregate. */

ahs_index++;
aggregate_type = DDIF$_LIN;
status = cda$create_aggregate( &root_aggregate_handle,              (41)
                               &aggregate_type,
                               &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/* Store the Polyline aggregate. */

aggregate_item = DDIF$_SEG_CONTENT;
status = cda$store_item( &root_aggregate_handle,                    (42)
                         &aggregate_handle_stack[ahs_index-1],
                         &aggregate_item,
                         &aggregate_handle_length,
                         &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/* Store Polyline Flags into the Polyline aggregate. */
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
aggregate_item = DDIF$_LIN_FLAGS;
local_length = sizeof( integer_value );
integer_value = 0x1;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &local_length,
                         &integer_value );
if( FAILURE( status ) )
    return ( status );

/* Store the Line Pattern bit string into the Polyline aggregate. */

aggregate_item = DDIF$_LIN_DRAW_PATTERN;
local_length = sizeof( integer_value );
integer_value = 0xF;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &local_length,
                         &integer_value );
if( FAILURE( status ) )
    return ( status );

/*
** For the points to be used, store "VALUE CONSTANT" as the data type
** choice, followed by the value of the point.
*/

for (i = 0; i < MAX_POINTS; i++ )
    {
    aggregate_item = DDIF$_LIN_PATH_C;
    local_length = sizeof( integer_value );
    integer_value = DDIF$K_VALUE_CONSTANT;
    aggregate_index = i * 2;
    status = cda$store_item( &root_aggregate_handle,          (43)
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &local_length,
                             &integer_value,
                             &aggregate_index );
    if( FAILURE( status ) )
        return ( status );

    /* Store the x-coordinate integer value in the polyline path array. */

    aggregate_item = DDIF$_LIN_PATH;
    local_length = sizeof( integer_value );
    integer_value = poly_points[i][0];
    aggregate_index = i * 2;
    status = cda$store_item( &root_aggregate_handle,          (44)
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &local_length,
                             &integer_value,
                             &aggregate_index );
    if( FAILURE( status ) )
        return ( status );
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
    /*
    ** Now store the y-coordinate for each point.
    */

    aggregate_item = DDIF$_LIN_PATH_C;
    local_length = sizeof( integer_value );
    integer_value = DDIF$K_VALUE_CONSTANT;
    aggregate_index = ((2 * i) + 1 );
    status = cda$store_item( &root_aggregate_handle,                    (45)
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &local_length,
                             &integer_value,
                             &aggregate_index );
    if( FAILURE( status ) )
        return ( status );

    aggregate_item = DDIF$_LIN_PATH;
    local_length = sizeof( integer_value );
    integer_value = poly_points[i][1];
    aggregate_index = ((2 * i) + 1 );
    status = cda$store_item( &root_aggregate_handle,                    (46)
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &local_length,
                             &integer_value,
                             &aggregate_index );
    if( FAILURE( status ) )
        return ( status );
    }; /* End of "for" loop */

ahs_index--; /* End of pline. */                                        (47)

/* Insert a new-page (hard) directive. */

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_hrd_dir ( &root_aggregate_handle,                      (48)
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_PAGE );
if( FAILURE( status ) )
    return ( status );

/* Insert next general-text line. */

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_gtx ( &root_aggregate_handle,                          (49)
                      &previous_aggregate_handle,
                      &aggregate_handle_stack[ahs_index],
                       "The following is a Bezier curve, using
                        the same path as the polyline, within a frame:" );
if( FAILURE( status ) )
    return ( status );

/* Insert two new-line directives to cause a skipped line. */
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_sft_dir ( &root_aggregate_handle,                    (50)
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
if( FAILURE( status ) )
    return ( status );

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_sft_dir ( &root_aggregate_handle,
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
if( FAILURE( status ) )
    return ( status );

/*
** Create new segment to define Bezier curve.
*/

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
aggregate_type = DDIF$_SEG;
status = cda$create_aggregate( &root_aggregate_handle,               (51)
                               &aggregate_type,
                               &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/*
** Insert after previous aggregate. (Insert rather than store, as
** this is a sequence of aggregates.)
*/

status = cda$insert_aggregate( &aggregate_handle_stack[ahs_index],
                               &previous_aggregate_handle );
if( FAILURE( status ) )
    return ( status );

/* Store the segment ID. */

aggregate_item = DDIF$_SEG_ID;
status = cda$store_item( &root_aggregate_handle,                     (52)
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &bline_label_length,
                          bline_label );
if( FAILURE( status ) )
    return ( status );

/* Store the segment type ("FRAME"). */

aggregate_item = DDIF$_SEG_SEGMENT_TYPE;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &tyd_label_length,
                          tyd_label );
if( FAILURE( status ) )
    return ( status );

/* Create a Bezier Curve aggregate. */
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
aggregate_type = DDIF$_BEZ;
previous_aggregate_handle = aggregate_handle_stack[ahs_index];
ahs_index++;
status = cda$create_aggregate( &root_aggregate_handle,          (53)
                               &aggregate_type,
                               &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/* Store the Bezier Curve aggregate */

aggregate_item = DDIF$_SEG_CONTENT;
status = cda$store_item( &root_aggregate_handle,                (54)
                         &aggregate_handle_stack[ahs_index-1],
                         &aggregate_item,
                         &aggregate_handle_length,
                         &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/* Store the Flags item into the Bezier Curve aggregate. */

aggregate_item = DDIF$_BEZ_FLAGS;
local_length = sizeof( integer_value );
integer_value = 0x1;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &local_length,
                         &integer_value );
if( FAILURE( status ) )
    return ( status );

/*
** For the points to be used, store "VALUE CONSTANT" as the data type
** choice, followed by the value of the point.
*/

for (i = 0; i < MAX_POINTS; i++ )
    {
    aggregate_item = DDIF$_BEZ_PATH_C;
    local_length = sizeof( integer_value );
    integer_value = DDIF$K_VALUE_CONSTANT;
    aggregate_index = i * 2;
    status = cda$store_item( &root_aggregate_handle,            (55)
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &local_length,
                             &integer_value,
                             &aggregate_index );
    if( FAILURE( status ) )
        return ( status );

    /* Store the x-coordinate integer value in the polyline path array. */
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
    aggregate_item = DDIF$_BEZ_PATH;
    local_length = sizeof( integer_value );
    integer_value = poly_points[i][0];
    aggregate_index = i * 2;
    status = cda$store_item( &root_aggregate_handle,                    (56)
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &local_length,
                             &integer_value,
                             &aggregate_index );
    if( FAILURE( status ) )
        return ( status );

    /*
    ** Now store the y-coordinate for each point.
    */

    aggregate_item = DDIF$_BEZ_PATH_C;
    local_length = sizeof( integer_value );
    integer_value = DDIF$K_VALUE_CONSTANT;
    aggregate_index = ((2 * i) + 1 );
    status = cda$store_item( &root_aggregate_handle,                    (57)
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &local_length,
                             &integer_value,
                             &aggregate_index );
    if( FAILURE( status ) )
        return ( status );

    aggregate_item = DDIF$_BEZ_PATH;
    local_length = sizeof( integer_value );
    integer_value = poly_points[i][1];
    aggregate_index = ((2 * i) + 1 );
    status = cda$store_item( &root_aggregate_handle,                    (58)
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &local_length,
                             &integer_value,
                             &aggregate_index );
    if( FAILURE( status ) )
        return ( status );
    }; /* End of "for" loop */

ahs_index--; /* End of Bezier segment */

/* Insert two new-line directives to cause a skipped line. */

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_sft_dir ( &root_aggregate_handle,                       (59)
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
if( FAILURE( status ) )
    return ( status );
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_sft_dir ( &root_aggregate_handle,
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
if( FAILURE( status ) )
    return ( status );

/* Insert next general-text line. */

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_gtx ( &root_aggregate_handle,                    (60)
                      &previous_aggregate_handle,
                      &aggregate_handle_stack[ahs_index],
                       "The following is a basketweave-filled arc
                        within a frame:" );
if( FAILURE( status ) )
    return ( status );

/* Insert two new-line directives to cause a skipped line. */

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_sft_dir ( &root_aggregate_handle,                (61)
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
if( FAILURE( status ) )
    return ( status );

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_sft_dir ( &root_aggregate_handle,
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
if( FAILURE( status ) )
    return ( status );

/*
** Create new segment to define special segment attributes for
** the arc.
*/

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
aggregate_type = DDIF$_SEG;
status = cda$create_aggregate( &root_aggregate_handle,           (62)
                               &aggregate_type,
                               &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/* Insert after previous aggregate. */

status = cda$insert_aggregate( &aggregate_handle_stack[ahs_index],
                               &previous_aggregate_handle );
if( FAILURE( status ) )
    return ( status );

/* Store the segment ID. */
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
aggregate_item = DDIF$_SEG_ID;
status = cda$store_item( &root_aggregate_handle,                    (63)
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &filled_arc_label_length,
                          filled_arc_label );
if( FAILURE( status ) )
    return ( status );

/* Store the segment type ("FRAME"). */

aggregate_item = DDIF$_SEG_SEGMENT_TYPE;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &tyd_label_length,
                          tyd_label );
if( FAILURE( status ) )
    return ( status );

/*
** Create a segment aggregate and store in the seg-content item.
*/

ahs_index++;
aggregate_type = DDIF$_SEG;
status = cda$create_aggregate( &root_aggregate_handle,              (64)
                               &aggregate_type,
                               &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_SEG_CONTENT;
status = cda$store_item( &root_aggregate_handle,                    (65)
                         &aggregate_handle_stack[ahs_index-1],
                         &aggregate_item,
                         &aggregate_handle_length,
                         &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/*
** Create new segment attributes aggregate to define the arc's
** attributes, and store it in the segment aggregate just created.
*/

ahs_index++;
aggregate_type = DDIF$_SGA;
status = cda$create_aggregate( &root_aggregate_handle,              (66)
                               &aggregate_type,
                               &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
    aggregate_item = DDIF$_SEG_SPECIFIC_ATTRIBUTES;
    status = cda$store_item( &root_aggregate_handle,                    (67)
                             &aggregate_handle_stack[ahs_index-1],
                             &aggregate_item,
                             &aggregate_handle_length,
                             &aggregate_handle_stack[ahs_index] );
    if( FAILURE( status ) )
        return ( status );

    /*
    ** Now store the specific attributes for the arc.
    */

    aggregate_item = DDIF$_SGA_LIN_WIDTH_C;
    integer_value = DDIF$K_VALUE_CONSTANT;
    status = cda$store_item( &root_aggregate_handle,
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &integer_length,
                             &integer_value );
    if( FAILURE( status ) )
        return ( status );

    aggregate_item = DDIF$_SGA_LIN_WIDTH;
    integer_value = arc_line_width;
    status = cda$store_item( &root_aggregate_handle,
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &integer_length,
                             &integer_value );
    if( FAILURE( status ) )
        return ( status );

    aggregate_item = DDIF$_SGA_LIN_STYLE;
    integer_value = DDIF$K_SOLID_LINE_STYLE;
    status = cda$store_item( &root_aggregate_handle,
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &integer_length,
                             &integer_value );
    if( FAILURE( status ) )
        return ( status );

    aggregate_item = DDIF$_SGA_LIN_END_START;
    integer_value = DDIF$K_ROUND_LINE_END;
    status = cda$store_item( &root_aggregate_handle,
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &integer_length,
                             &integer_value );
    if( FAILURE( status ) )
        return ( status );
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
aggregate_item = DDIF$_SGA_LIN_END_FINISH;
integer_value = DDIF$K_ROUND_LINE_END;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_SGA_LIN_JOIN;
integer_value = DDIF$K_MITERED_LINE_JOIN;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_SGA_LIN_INTERIOR_PATTERN;
integer_value = DDIF$K_PATT_BASKET_WEAVE;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value );
if( FAILURE( status ) )
    return ( status );

ahs_index--; /* End of arc attributes */                              (68)

/* Create an Arc Content aggregate. */

aggregate_type = DDIF$_ARC;
previous_aggregate_handle = aggregate_handle_stack[ahs_index];
ahs_index++;
status = cda$create_aggregate( &root_aggregate_handle,               (69)
                               &aggregate_type,
                               &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/*
** Store the arc-content aggregate as the seg_content of the previous
** aggregate.
*/

aggregate_item = DDIF$_SEG_CONTENT;
status = cda$store_item( &root_aggregate_handle,                     (70)
                         &aggregate_handle_stack[ahs_index-1],
                         &aggregate_item,
                         &aggregate_handle_length,
                         &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/* Store the Flags item into the arc aggregate. */
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
set_arc_flags.ddif$v_arc_draw_arc = 1;
set_arc_flags.ddif$v_arc_fill_arc = 1;
set_arc_flags.ddif$v_arc_pie_arc = 1;
set_arc_flags.ddif$v_arc_close_arc = 0;
set_arc_flags.ddif$v_arc_flags_fill = 0;

aggregate_item = DDIF$_ARC_FLAGS;
local_length = sizeof( integer_value );
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &local_length,
                         &set_arc_flags );
if( FAILURE( status ) )
    return ( status );

/* Store "VALUE CONSTANT" as the data type choice for the arc
   center x-coordinate. */

aggregate_item = DDIF$_ARC_CENTER_X_C;
local_length = sizeof( integer_value );
integer_value = DDIF$K_VALUE_CONSTANT;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &local_length,
                         &integer_value );
if( FAILURE( status ) )
    return ( status );

/* Store an integer value for the arc center x-coordinate. */

aggregate_item = DDIF$_ARC_CENTER_X;
local_length = sizeof( integer_value );
integer_value = 3000;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &local_length,
                         &integer_value );
if( FAILURE( status ) )
    return ( status );

/* Store "VALUE CONSTANT" as the data type choice for the arc
   center y-coordinate. */

aggregate_item = DDIF$_ARC_CENTER_Y_C;
local_length = sizeof( integer_value );
integer_value = DDIF$K_VALUE_CONSTANT;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &local_length,
                         &integer_value );
if( FAILURE( status ) )
    return ( status );

/* Store an integer value for the arc center y-coordinate. */
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
aggregate_item = DDIF$_ARC_CENTER_Y;
local_length = sizeof( integer_value );
integer_value = 150;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &local_length,
                         &integer_value );
if( FAILURE( status ) )
    return ( status );

/* Store "VALUE CONSTANT" as the data type choice for the arc
   radius x value. */

aggregate_item = DDIF$_ARC_RADIUS_X_C;
local_length = sizeof( integer_value );
integer_value = DDIF$K_VALUE_CONSTANT;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &local_length,
                         &integer_value );
if( FAILURE( status ) )
    return ( status );

/* Store an integer value for the arc radius x value. */

aggregate_item = DDIF$_ARC_RADIUS_X;
local_length = sizeof( integer_value );
integer_value = 2000;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &local_length,
                         &integer_value );
if( FAILURE( status ) )
    return ( status );

/* Store "ANGLE CONSTANT" as the data type choice for the arc
   start value. */

aggregate_item = DDIF$_ARC_START_C;
local_length = sizeof( integer_value );
integer_value = DDIF$K_ANGLE_CONSTANT;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &local_length,
                         &integer_value );
if( FAILURE( status ) )
    return ( status );

/* Store an integer value for the arc start value. */
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
aggregate_item = DDIF$_ARC_START;
local_length = sizeof( arc_start );
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &local_length,
                         &arc_start );
if( FAILURE( status ) )
    return ( status );

/* Store "ANGLE CONSTANT" as the data type choice for the arc
   EXTENT value. */

aggregate_item = DDIF$_ARC_EXTENT_C;
local_length = sizeof( integer_value );
integer_value = DDIF$K_ANGLE_CONSTANT;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &local_length,
                         &integer_value );
if( FAILURE( status ) )
    return ( status );

/* Store an integer value for the arc EXTENT value. */

aggregate_item = DDIF$_ARC_EXTENT;
local_length = sizeof( arc_extent );
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &local_length,
                         &arc_extent );
if( FAILURE( status ) )
    return ( status );

ahs_index--; /* End of arc. */
ahs_index--; /* End of arc-attribute segment */

/* Insert two new-line directives to cause a skipped line. */

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_sft_dir ( &root_aggregate_handle,                (71)
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
if( FAILURE( status ) )
    return ( status );

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_sft_dir ( &root_aggregate_handle,
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
if( FAILURE( status ) )
    return ( status );

/* Insert next general-text line. */
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
    previous_aggregate_handle = aggregate_handle_stack[ahs_index];
    status = create_gtx ( &root_aggregate_handle,                    (72)
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           "This ends the examples." );
    if( FAILURE( status ) )
        return ( status );

    ahs_index--; /* End of image segment */
    ahs_index--; /* End of document content. */

    /* Create an output file to receive the DDIF stream. */

    status = cda$create_file( &filename_length,
                              filename,
                              0, 0, 0, 0, 0,
                             &root_aggregate_handle,
                             &result_file_spec_len,
                              result_file_spec,
                             &result_file_spec_len,
                             &stream_handle,
                             &file_handle,
                             &root_aggregate_handle );
    if( FAILURE( status ) )
        return ( status );

    result_file_spec[result_file_spec_len] = 0;
    printf("Created file: %s\n",result_file_spec );

    /* Write the DDIF stream to the output file */

    printf("Writing document...\n" );

    status = cda$put_document( &root_aggregate_handle,
                               &stream_handle );
    if( FAILURE( status ) )
        return ( status );

    /* Close the output file. */

    status = cda$close_file( &stream_handle,
                            &file_handle );
    if( FAILURE( status ) )
        return ( status );

    /* Delete the Root aggregate structure. */

    status = cda$delete_root_aggregate( &root_aggregate_handle );
    if( FAILURE( status ) )
        return ( status );

    return;
}
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
/*
** This routine creates a soft-directive aggregate for the specified
** directive type, and inserts it after the specified previous
** aggregate. It returns the handle of the newly-created aggregate.
*/

unsigned long create_sft_dir (root_handle,
                              previous_handle,
                              return_handle,
                              dir_type )

unsigned long *root_handle;                    /* Root aggregate handle. */
unsigned long *previous_handle;                /* previous handle */
unsigned long *return_handle;                  /* Handle to be returned. */
unsigned long dir_type;                        /* Directive item code. */

{
unsigned long   aggregate_handle;
unsigned long   aggregate_handle_length = sizeof( aggregate_handle );
unsigned long   aggregate_type;
unsigned long   aggregate_item;
unsigned long   integer_value;
unsigned long   local_length;
unsigned long   status;

    /* Create a Soft Directive aggregate. */

    aggregate_type = DDIF$_SFT;
    status = cda$create_aggregate(root_handle,
                                  &aggregate_type,
                                  &aggregate_handle );
    if( FAILURE( status ) )
        return ( status );

    /* Insert the Soft Directive aggregate in the sequence of aggregates. */

    status = cda$insert_aggregate( &aggregate_handle,
                                    previous_handle );
    if( FAILURE( status ) )
        return ( status );

    /* Store the designated directive as an item in the
       Soft Directive aggregate. */

    aggregate_item = DDIF$_SFT_DIRECTIVE;
    local_length = sizeof( integer_value );
    integer_value = dir_type;
    status = cda$store_item(root_handle,
                            &aggregate_handle,
                            &aggregate_item,
                            &local_length,
                            &integer_value );
    if( FAILURE( status ) )
        return ( status );

    *return_handle = aggregate_handle;

    return(1 );
}
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
/*
** This routine creates a hard-directive aggregate for the specified
** directive type, and inserts it after the specified previous
** aggregate. It returns the handle of the newly-created aggregate.
*/

unsigned long create_hrd_dir (root_handle,
                              previous_handle,
                              return_handle,
                              dir_type )

unsigned long *root_handle;                    /* Root aggregate handle. */
unsigned long *previous_handle;                /* previous handle */
unsigned long *return_handle;                  /* Handle to be returned. */
unsigned long dir_type;                        /* Directive item code. */

{
unsigned long   aggregate_handle;
unsigned long   aggregate_handle_length = sizeof( aggregate_handle );
unsigned long   aggregate_type;
unsigned long   aggregate_item;
unsigned long   integer_value;
unsigned long   local_length;
unsigned long   status;

    /* Create a Hard Directive aggregate. */

    aggregate_type = DDIF$_HRD;
    status = cda$create_aggregate(root_handle,
                                  &aggregate_type,
                                  &aggregate_handle );
    if( FAILURE( status ) )
        return ( status );

    /* Insert the Hard Directive aggregate in the sequence of aggregates. */

    status = cda$insert_aggregate( &aggregate_handle,
                                    previous_handle );
    if( FAILURE( status ) )
        return ( status );

    /* Store the designated directive as an item in the
       Hard Directive aggregate. */

    aggregate_item = DDIF$_HRD_DIRECTIVE;
    local_length = sizeof( integer_value );
    integer_value = dir_type;
    status = cda$store_item(root_handle,
                            &aggregate_handle,
                            &aggregate_item,
                            &local_length,
                            &integer_value );
    if( FAILURE( status ) )
        return ( status );

    *return_handle = aggregate_handle;

    return(1 );
}
```

(continued on next page)

Example B–1 (Cont.) Sample CDA Toolkit Program

```
/*
** This routine creates a general-text aggregate for the specified
** text, and inserts it after the specified previous aggregate. It
** returns the handle of the newly-created aggregate.
*/

unsigned long create_gtx (root_handle,
                          previous_handle,
                          return_handle,
                          gtx_string )

unsigned long *root_handle;                    /* Root aggregate handle. */
unsigned long *previous_handle;                /* previous handle */
unsigned long *return_handle;                  /* Handle to be returned. */
char          *gtx_string;                     /* Ptr to text string. */

{
unsigned long   aggregate_handle;
unsigned long   aggregate_handle_length = sizeof( aggregate_handle );
unsigned long   aggregate_type;
unsigned long   aggregate_item;
unsigned long   add_info;
unsigned long   integer_value;
unsigned long   local_length;
unsigned long   status;

    /* Create another General Text Content aggregate. */

    aggregate_type = DDIF$_GTX;
    status = cda$create_aggregate(root_handle,
                                  &aggregate_type,
                                  &aggregate_handle );
    if( FAILURE( status ) )
        return ( status );

    /* Insert the Text aggregate in the sequence of aggregates. */

    status = cda$insert_aggregate( &aggregate_handle,
                                    previous_handle );
    if( FAILURE( status ) )
        return ( status );

    /* Store more text into the General Text aggregate. */

    aggregate_item = DDIF$_GTX_CONTENT;
    add_info = CDA$K_ISO_LATIN1;
    local_length = strlen( gtx_string );
    status = cda$store_item(root_handle,
                            &aggregate_handle,
                            &aggregate_item,
                            &local_length,
                             gtx_string,
                             0,
                            &add_info );
    if( FAILURE( status ) )
        return ( status );

    *return_handle = aggregate_handle;

    return(1 );
}
```

Example B–2 illustrates the Analysis output of the DDIF document created by Example B–1. The callouts in Example B–1 correspond to the callouts in Example B–2.

In the Analysis output of a DDIF file, the following symbols are used.

- A left brace indicates the beginning of an aggregate.
- A right brace indicates the end of an aggregate.
- A left parenthesis indicates the beginning of an array.
- A right parenthesis indicates the end of an array.

Additionally, in this example all hexadecimal values produced by the Analysis back end have been restored to their ASCII representations.

Note that default values are indicated by the comment "[Default value.]". These values are not specified in Example B–1; instead, the default values specified by the CDA Toolkit are accepted.

Example B–2 Analysis Output of DDIF File

```
  DDIF_DOCUMENT
❶ {
❸  DDF_DESCRIPTOR
❷  (
❹   DSC_MAJOR_VERSION  1   ! Longword Integer
    DSC_MINOR_VERSION  0   ! Longword Integer
    DSC_PRODUCT_IDENTIFIER  "DDIF$"
❺   DSC_PRODUCT_NAME
    (
     ISO_LATIN1  "Test V1.0"
    )
   }
❼  DDF_HEADER
❻  (
    DHD_LANGUAGES_C
    (
❽    ISO_639_LANGUAGE  ! Integer = 0
❿    OTHER_LANGUAGE  ! Integer = 1
    )
    DHD_LANGUAGES
    (
❾    "E/USA/"
⓫    ISO_LATIN1  "Mandarin"
    )
   }
⓭  DDF_CONTENT
⓬  (
⓯   SEG_SPECIFIC_ATTRIBUTES
```

(continued on next page)

Example B–2 (Cont.) Analysis Output of DDIF File

```
14  {
16   SGA_CONTENT_CATEGORY  T_CATEGORY  "$T"
    }
18  SEG_CONTENT
17  {
19   TXT_CONTENT  "This is the first line of the example text."
    }
20  {
    HRD_DIRECTIVE  DIR_SPACE  ! Integer = 5
    }
21  {
     GTX_CONTENT  ISO_LATIN1  "This is the second line of the example text,
  and should be separated from the first line by a single space."  ! Char. string.
    }
22  {
     SFT_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
    }
23  {
     GTX_CONTENT  ISO_LATIN1  "The third line of the example text will begin
  on a new line."  ! Char. string.
    }
24  {
     SFT_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
    }
    {
     SFT_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
    }
25  {
     GTX_CONTENT  ISO_LATIN1  "The fourth line of the example text will be
  separated from the previous lines by a blank line." ! Char. string.
    }
26  {
     SFT_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
    }
    {
     SFT_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
    }
27  {
     GTX_CONTENT  ISO_LATIN1  "The following is a polyline within a frame:"
  ! Char. string.
    }
28  {
     SFT_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
    }
    {
     SFT_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
    }
29  {
31   SEG_SPECIFIC_ATTRIBUTES
30   {
33    SGA_TYPE_DEFNS
32    {
34     TYD_LABEL  "FRAME"
36     TYD_ATTRIBUTES
35     {
37      SGA_CONTENT_CATEGORY  TWOD_CATEGORY  "$2D"
        SGA_FRM_FLAGS  "%B01000000000000000000000000000000"  ! Flags
```

(continued on next page)

Example B–2 (Cont.) Analysis Output of DDIF File

```
        SGA_FRM_BOX_LL_X_C  VALUE_CONSTANT  ! Integer = 0
        SGA_FRM_BOX_LL_X  0  ! Longword Integer
        SGA_FRM_BOX_LL_Y_C  VALUE_CONSTANT  ! Integer = 0
        SGA_FRM_BOX_LL_Y  0  ! Longword Integer
        SGA_FRM_BOX_UR_X_C  VALUE_CONSTANT  ! Integer = 0
        SGA_FRM_BOX_UR_X  6000  ! Longword Integer
        SGA_FRM_BOX_UR_Y_C  VALUE_CONSTANT  ! Integer = 0
        SGA_FRM_BOX_UR_Y  2400  ! Longword Integer
        SGA_FRM_POSITION_C  FRAME_GALLEY  ! Integer = 2
        SGA_FRMGLY_VERTICAL  FRMGLY_BELOW_CURRENT  ! Integer = 1  [Default value.]
        SGA_FRMGLY_HORIZONTAL  FMT_CENTER_OF_PATH  ! Integer = 2  [Default value.]
      }
     }
    }
39  SEG_CONTENT
38  {
40   SEG_ID  "pline"
     SEG_SEGMENT_TYPE  "FRAME"
42   SEG_CONTENT
41   {
      LIN_FLAGS  "%B10000000000000000000000000000000"  ! Flags
      LIN_DRAW_PATTERN  "%B1111"  ! Bit string
      LIN_PATH_C
      (
43     VALUE_CONSTANT  ! Integer = 0
45     VALUE_CONSTANT  ! Integer = 0
       VALUE_CONSTANT  ! Integer = 0
       VALUE_CONSTANT  ! Integer = 0
       VALUE_CONSTANT  ! Integer = 0
       VALUE_CONSTANT  ! Integer = 0
       VALUE_CONSTANT  ! Integer = 0
       VALUE_CONSTANT  ! Integer = 0
      )
      LIN_PATH
      (
44     500  ! Integer
46     500  ! Integer
       2500  ! Integer
       2000  ! Integer
       3500  ! Integer
       2000  ! Integer
       5500  ! Integer
       500  ! Integer
      )
     }
47  }
48  {
     HRD_DIRECTIVE  DIR_NEW_PAGE  ! Integer = 1
    }
49  {
     GTX_CONTENT  ISO_LATIN1  "The following is a Bezier curve, using the
same path as the polyline, within a frame:"  ! Char. string.
    }
50  {
     SFT_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
    }
    {
```

(continued on next page)

Example B–2 (Cont.) Analysis Output of DDIF File

```
      SFT_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
     }
51   {
52    SEG_ID  "bline"
      SEG_SEGMENT_TYPE  "FRAME"
54    SEG_CONTENT
53    {
       BEZ_FLAGS  "%B10000000000000000000000000000000"  ! Flags
       BEZ_PATH_C
       (
55      VALUE_CONSTANT  ! Integer = 0
57      VALUE_CONSTANT  ! Integer = 0
        VALUE_CONSTANT  ! Integer = 0
        VALUE_CONSTANT  ! Integer = 0
        VALUE_CONSTANT  ! Integer = 0
        VALUE_CONSTANT  ! Integer = 0
        VALUE_CONSTANT  ! Integer = 0
        VALUE_CONSTANT  ! Integer = 0
       )
       BEZ_PATH
       (
56      500  ! Integer
58      500  ! Integer
        2500  ! Integer
        2000  ! Integer
        3500  ! Integer
        2000  ! Integer
        5500  ! Integer
        500  ! Integer
       )
      }
     }
59   {
      SFT_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
     }
     {
      SFT_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
     }
60   {
      GTX_CONTENT  ISO_LATIN1  "The following is a basketweave-filled arc
  within a frame:"  ! Char. string.
     }
61   {
      SFT_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
     }
     {
      SFT_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
     }
62   {
63    SEG_ID  "filled_arc"
      SEG_SEGMENT_TYPE  "FRAME"
65    SEG_CONTENT
64    {
67     SEG_SPECIFIC_ATTRIBUTES
66     {
        SGA_LIN_WIDTH_C  VALUE_CONSTANT  ! Integer = 0
        SGA_LIN_WIDTH  60  ! Longword Integer
```

(continued on next page)

Example B–2 (Cont.) Analysis Output of DDIF File

```
        SGA_LIN_STYLE  SOLID_LINE_STYLE  ! Integer = 1
        SGA_LIN_END_START  ROUND_LINE_END  ! Integer = 2
        SGA_LIN_END_FINISH  ROUND_LINE_END  ! Integer = 2
        SGA_LIN_JOIN  MITERED_LINE_JOIN  ! Integer = 1
        SGA_LIN_INTERIOR_PATTERN  41  ! Longword Integer
68     }
70     SEG_CONTENT
69     {
        ARC_FLAGS  "%B11100000000000000000000000000000"  ! Flags
        ARC_CENTER_X_C  VALUE_CONSTANT  ! Integer = 0
        ARC_CENTER_X  3000  ! Longword Integer
        ARC_CENTER_Y_C  VALUE_CONSTANT  ! Integer = 0
        ARC_CENTER_Y  150  ! Longword Integer
        ARC_RADIUS_X_C  VALUE_CONSTANT  ! Integer = 0
        ARC_RADIUS_X  2000  ! Longword Integer
        ARC_RADIUS_DELTA_Y_C  VALUE_CONSTANT  ! Integer = 0  [Default value.]
        ARC_RADIUS_DELTA_Y  0  ! Longword Integer  [Default value.]
        ARC_START_C  ANGLE_CONSTANT  ! Integer = 0
        ARC_START "%F4.500000e+01"  ! Single Prec. Floating Point
        ARC_EXTENT_C  ANGLE_CONSTANT  ! Integer = 0
        ARC_EXTENT "%F9.000000e+01"  ! Single Prec. Floating Point
        ARC_ROTATION_C  ANGLE_CONSTANT  ! Integer = 0  [Default value.]
        ARC_ROTATION "%F0.000000e+00"  ! Single Prec. Floating Point  [Default value.]
       }
      }
     }
71   {
      SFT_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
     }
     {
      SFT_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
     }
72   {
      GTX_CONTENT  ISO_LATIN1  "This ends the examples."  ! Char. string.
     }
    }
   }
  }
```

C Text Front End Source File

This appendix contains the source code for the Text front end provided with the CDA Toolkit. This front end should be used as a sample when writing your own front or back ends. The Text front end reads in a standard text file and creates a DDIF in-memory document.

In this appendix, the source code for the Text front end is divided into subsections. Where appropriate, the subsections are annotated with a list following each section explaining the annotations.

The following callouts correspond to the callouts in the main module of the Text front end.

❶ All of these routines from the CDA Toolkit are used by the Text front end.

❷ These are the additional entry points in the Text front end.

❸ This is the context block that is used to share information between the front end, the CDA Converter Kernel, and the back end.

```
/*
**++
**
**  COPYRIGHT (c) 1987 BY
**  DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
**  ALL RIGHTS RESERVED.
**
**  THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND  COPIED
**  ONLY  IN  ACCORDANCE  WITH  THE  TERMS  OF  SUCH  LICENSE AND WITH THE
**  INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR  ANY  OTHER
**  COPIES  THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
**  OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE  IS  HEREBY
**  TRANSFERRED.
**
**  THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE  WITHOUT  NOTICE
**  AND  SHOULD  NOT  BE  CONSTRUED  AS  A COMMITMENT BY DIGITAL EQUIPMENT
**  CORPORATION.
**
**  DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR  RELIABILITY  OF  ITS
**  SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
**
**  FACILITY:
**
**      Compound Document Converters
**
**  ABSTRACT:
**
**      This is a Text Converter Front End that reads a text input
**      file (or stream), creates DDIF aggregates from this text, and
**      passes each DDIF Aggregate back to the calling converter kernel.
**
**--
**/

/*
**
**  INCLUDE FILES
**
**/

#include <ddif$def.h>     /* Contains values of al DDIF$xxxx keywords */
#include <cda$def.h>      /* Contains values of all CDA$xxxx keywords */
#include <cda$msg.h>      /* CDA error messages */
```

Text Front End Source File

```
#ifdef vms           /* Use VMS RMS to manipulate files */
#include <fab.h>     /* Defines the file access block structure */
#include <rab.h>     /* Defines the record access block structure */
#include <nam.h>     /* Defines the name block structure */
#include <rmsdef.h>  /* Defines the completion status codes that RMS returns
                      *  after every file- or record-processing operation */

/* NOTE: The pervious 4 #include statements can be replaced with <rms.h> */

#include <descrip.h> /* Allows program to pass arguments by descriptor.
                      * A descriptor is a structure that describes the
                      * data type, size, and address of a data structure. */
#endif

/* Declare routines used in the Toolkit */                              ❶

extern unsigned long cda$open_text_file();
extern unsigned long cda$close_text_file();
extern unsigned long cda$read_text_file();
extern unsigned long cda$get_aggregate();
extern unsigned long cda$get_text_position();
extern unsigned long cda$create_root_aggregate();
extern unsigned long cda$delete_root_aggregate();
extern unsigned long cda$create_aggregate();
extern unsigned long cda$store_item();

unsigned long get_aggregate();                                          ❷
unsigned long create_dsc();
unsigned long create_dhd();
unsigned long create_seg();
unsigned long create_txt();
unsigned long create_eos();
unsigned long look_ahead();
unsigned long create_dir();
unsigned long get_position();
unsigned long close_front_end();

/* Define literals for characters used */
#define HORIZONTAL_TAB  9
#define FORM_FEED  12
#define DDIF_BUFFER_SIZE 2048

/* Front End Context structure (text context)
 * The front end context contains all variables needed to keep track
 * of a conversion in progress. Since the front end, back end, and
 * converter kernel are re-entrant, it is possible to have several
 * conversions occurring simultaneously. A pointer to this structure
 * is passed back and forth between the front and back ends, so
 * that the front end knows where it is in any particular conversion.
 */
struct text_cxt {                                                       ❸
        unsigned long text_a_file_handle;
        unsigned long text_a_root_aggregate_handle;
        unsigned long (*text_a_input_routine)();
        unsigned long text_a_input_routine_param;
        unsigned long (*text_a_position_routine)();
        unsigned long text_a_position_param;
        unsigned long text_l_state;
        unsigned char *text_a_buffer_address;
        unsigned long text_l_buffer_length;
        unsigned char *text_a_local_buffer;
        unsigned char text_l_local_length;
        unsigned long text_l_directive_type;
        unsigned long text_l_directive_content;
        unsigned char text_a_title[32];
        unsigned long text_l_title_length;
        unsigned long text_b_scope_level;
        unsigned long text_l_newline_count;
        unsigned char text_v_end_of_paragraph : 1;
        unsigned char text_v_root_segment     : 1;
        unsigned char text_v_end_of_document  : 1;
        unsigned char : 0;
};

/* Default file extension */
static unsigned char    default_file[] = ".txt";
static unsigned long    default_length = sizeof(default_file) - 1;
```

```
/* Name for Root Segment */
static unsigned char    seg_id[] = "RootSegment";
static unsigned long    seg_id_length = sizeof(seg_id) - 1;

/* Name for style guide file */
static unsigned char    style_guide_name[] = "defstyle";
static unsigned long    style_length = sizeof(style_guide_name) - 1;

/* Name for paragraph */
static unsigned char    para_buffer[] = "PARA";
static unsigned long    para_length = sizeof(para_buffer) - 1;

/* Name for literal */
static unsigned char    literal_buffer[] = "LITERAL";
static unsigned long    literal_length = sizeof(literal_buffer) - 1;

/* Name for erf descriptor */
static unsigned char    erf_desc_type[] = "Style Guide";
static unsigned long    erf_desc_length = sizeof(erf_desc_type) - 1;

/* Name for erf label type */
static unsigned char    erf_label_type[] = "$STYLE";
static unsigned long    erf_length = sizeof(erf_label_type) - 1;

/*
**
**  MACROS
**
**/

/* Error check macros */

#define FAILURE(status)                                         \
        (((status) & 1) == 0)

#define SUCCESS(status)                                         \
        (((status) & 1) == 1)

/* Memory allocation and deallocation */

#ifdef vms
extern unsigned long lib$free_vm();
extern unsigned long lib$get_vm();
#else
extern char *malloc();
extern free();
#endif

/* Literals used in creation of aggregates */

static unsigned char  dsc_identifier[] = "DDIF$";
static unsigned long  dsc_id_length    = sizeof(dsc_identifier) - 1;
static unsigned char  dsc_prod_name[]  = "DDIF Text Front End";
static unsigned long  dsc_nam_length   = sizeof(dsc_prod_name) - 1;
static unsigned char  dhd_author[]     = "DDIF Text Front End";
static unsigned long  dhd_aut_length   = sizeof(dhd_author) - 1;

/* Lookup table for DEC MCS character set.  These values are taken from DEC
 * Standard 169. This table has the space character inserted in the control
 * character and holes positions.  This ensures no such characters appear
 * in the DDIF TXT aggregates.
 */
static unsigned char lookup_buffer[256] =
{32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,
 32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,
 32,  33,  34,  35,  36,  37,  38,  39,  40,  41,  42,  43,  44,  45,  46,  47,
 48,  49,  50,  51,  52,  53,  54,  55,  56,  57,  58,  59,  60,  61,  62,  63,
 64,  65,  66,  67,  68,  69,  70,  71,  72,  73,  74,  75,  76,  77,  78,  79,
 80,  81,  82,  83,  84,  85,  86,  87,  88,  89,  90,  91,  92,  93,  94,  95,
 96,  97,  98,  99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111,
112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126,  32,

 32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,
 32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,  32,
 32, 161, 162, 163,  32, 165,  32, 167, 168, 169, 170, 171,  32,  32,  32,  32,
176, 177, 178, 179,  32, 181, 182, 183,  32, 185, 186, 187, 188, 189,  32, 191,
192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207,
208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221,  32, 223,
224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239,
240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253,  32,  32};
```

The following callout corresponds to the callout in the jacket entry point for the Text front end.

❹ This is a jacket routine that supports the ULTRIX entry point to the Text front end.

```
/*
**++
**  FUNCTIONAL DESCRIPTION:                                              ❹
**
**      The name of this routine is CDA$READ_FORMAT().
**      This routine is the jacket entry point for the text Front End on
**      Ultrix and OS/2.  It is called from the converter kernel to
**      call the "real" entry point which initializes the conversion.
**      When employed on VMS systems, this routine is not called (or even
**      compiled). On VMS systems, the converter kernel calls the
**      DDIF$READ_TEXT() routine.
**
**  FORMAL PARAMETERS:
**
**      item_list.rr.ra          item list
**
**      cvt_context.rlu.v        value for cda$open_converter
**
**      text_context.wlu.v       value to identify this converter
**
**      get_aggr.wa.r            address of get aggregate routine
**
**      get_pos.wa.r             address of get position routine
**
**      close_text.wa.r          address of close front end routine
**
**  IMPLICIT INPUTS:
**
**      none
**
**  IMPLICIT OUTPUTS:
**
**      none
**
**  FUNCTION VALUE:
**
**      CDA$_NORMAL
**      CDA$_INVAGGTYP
**      Memory allocation error conditions
**      File error conditions
**
**  SIDE EFFECTS:
**
**      none
**
**--
**/
#ifndef vms
unsigned long   cda$read_format(item_list,
                                cvtr_context,
                                text_context_ptr,
                                get_aggr,
                                get_pos,
                                close_text)
```

```
struct item_list        *item_list;
unsigned long           cvtr_context;
unsigned long           *text_context_ptr;
unsigned long           *get_aggr;
unsigned long           *get_pos;
unsigned long           *close_text;

{
unsigned long ddif$read_text();

         return (ddif$read_text(item_list, cvtr_context, text_context_ptr,
                                get_aggr, get_pos, close_text));
}
#endif
```

The following callouts correspond to the callouts in the main entry point of the Text front end.

❺ This is the main entry point of the Text front end.

❻ This loop reads the items in the item list passed to the Text front end. This item list can contain information such as the file specification of the file to be used for input, the routine to be used to read the input, a parameter to the input routine, and so on.

❼ This statement creates the DDIF root aggegate (type DDIF$_DDF). This aggregate is required in every DDIF document.

❽ The next aggregate to be created is the document descriptor aggregate (type DDIF$_DSC). This aggregate is also required in every DDIF document.

```
/*
**++
**  FUNCTIONAL DESCRIPTION:                                              ❺
**
**      This routine is the entry point for the Text Front End.  It
**      is called from the converter kernel to initialize the
**      conversion.
**
**  GENERAL DESCRIPTION:
**
**      The DDIF$READ_format entry point is the initial entry point in the
**      front end. This routine initializes the conversion process and
**      establishes any special processing information for the front end.
**      The term "format" in the entry point name refers to the name of the
**      document format that is read by this particular front end ---
**      "TEXT", in this instance.
**
**      This routine is required and must be named according to the above
**      convention. Three other routines/entry points are also required.
**      The parameters to this routine specify their addresses to the
**      converter kernel.
**
```

```
**  FORMAL PARAMETERS:
**
**      item_list.rr.ra          item list
**
**      cvt_context.rlu.v        value for cda$open_converter
**
**      text_context.wlu.v       value to identify this converter
**
**  The next three parameters are the addresses of the other required
**  entry points in any front end.
**
**      get_aggr.wa.r            address of get aggregate routine
**
**      get_pos.wa.r             address of get position routine
**
**      close_text.wa.r          address of close front end routine
**
**  IMPLICIT INPUTS:
**
**      text file or data stream
**
**  IMPLICIT OUTPUTS:
**
**      none
**
**  FUNCTION VALUE:
**
**      CDA$_NORMAL
**      CDA$_INVAGGTYP
**      Memory allocation error conditions
**      File error conditions
**
**  SIDE EFFECTS:
**
**      none
**
**--
**/
unsigned long   ddif$read_text (item_list,
                                cvtr_context,
                                text_context_ptr,
                                get_aggr,
                                get_pos,
                                close_text)

struct item_list        *item_list;
unsigned long           cvtr_context;
unsigned long           *text_context_ptr;
unsigned long           *get_aggr;
unsigned long           *get_pos;
unsigned long           *close_text;

{
unsigned long   status;                 /* return status */
unsigned long   struct_size;            /* holds context block size */
unsigned long   aggregate_type;         /* aggregate type*/
unsigned long   result_length;          /* result file length */
unsigned char   result_buffer[255];     /* result file buffer */
unsigned long   filespec_length;        /* file specification length */
unsigned char   *default_file_address;
unsigned long   default_file_length;
unsigned char   *input_file_address;
unsigned long   input_file_length;
struct text_cxt *text_context;          /* points to context block */
```

```
        /* Allocate the context block for this front end */
        struct_size = sizeof (struct text_cxt);
        text_context = 0;
        default_file_address = default_file;
        default_file_length  = default_length;
        input_file_address   = 0;
        input_file_length    = 0;

#ifdef vms
        status = lib$get_vm(&struct_size, &text_context, 0);
#else
        text_context = (struct text_cxt *) malloc(struct_size);
        (text_context == 0) ? (status = CDA$_ALLOCFAIL) : (status = 1);
#endif
        if (FAILURE(status))
                return (status);

        /* Initialize the context block */
        text_context->text_a_file_handle           = 0;
        text_context->text_a_root_aggregate_handle  = 0;
        text_context->text_a_input_routine         = 0;
        text_context->text_a_input_routine_param   = 0;
        text_context->text_a_position_routine      = 0;
        text_context->text_a_position_param        = 0;
        text_context->text_l_state                 = 0;
        text_context->text_l_title_length          = 0;
        text_context->text_a_buffer_address        = 0;
        text_context->text_l_buffer_length         = 0;
        text_context->text_a_local_buffer          = 0;
        text_context->text_l_local_length          = 0;
        text_context->text_l_directive_type        = 0;
        text_context->text_l_directive_content     = 0;
        text_context->text_b_scope_level           = 0;
        text_context->text_l_newline_count         = 0;
        text_context->text_v_root_segment          = 1;
        text_context->text_v_end_of_paragraph      = 0;
        text_context->text_v_end_of_document       = 0;

        /* Scan item list until item code is 0 */
        while (item_list->cda$w_item_code != 0)                    ❻
        {
                status = 1;
                switch (item_list->cda$w_item_code)
                {
                  case CDA$_INPUT_FILE:        /* Input filename */
                      input_file_length  = item_list->cda$w_item_length;
                      input_file_address = (unsigned char *)
                                           item_list->cda$a_item_address;
                      break;

                  case CDA$_INPUT_DEFAULT:     /* Default input filename */
                      default_file_length  = item_list->cda$w_item_length;
                      default_file_address = (unsigned char *)
                                             item_list->cda$a_item_address;
                      break;

                  case CDA$_INPUT_PROCEDURE:   /* Input procedure address */
                      text_context->text_a_input_routine =
                                           (unsigned long (*)())
                                           item_list->cda$a_item_address;
                      break;
```

```
                case CDA$_INPUT_PROCEDURE_PARM: /* Input procedure param */
                    text_context->text_a_input_routine_param =
                                    *((unsigned long *)
                                    item_list->cda$a_item_address);
                    break;

                case CDA$_INPUT_POSITION_PROCEDURE: /* Input position
                                                       proc address */
                    text_context->text_a_position_routine =
                                    (unsigned long (*)())
                                    item_list->cda$a_item_address;
                    break;

                 default:                          /* All others */
                    break;
              }

              /* Any problems? */
              if (FAILURE(status))
                      return (status);

            /* Point to next item in item list                              */
            /* Note that this advances the item list a full two longwords */
            /* (i.e. + 1 * sizeof(item_list))                               */
              item_list += 1;
        }

        /* Create a DDIF root aggregate */
        aggregate_type = DDIF$_DDF;                                          ❼
        status = cda$create_root_aggregate (0,
                            0,
                            0,
                            0,
                            &aggregate_type,
                            &text_context->text_a_root_aggregate_handle);

        /* If there is an error, deallocate context block and return */
        if (FAILURE(status))
                return (status);

        /* Try to open the input file if specified */
        if (input_file_address != 0)
        {
                result_length  = sizeof (result_buffer);

                status = cda$open_text_file (&input_file_length,
                                             input_file_address,
                                             &default_file_length,
                                             default_file_address,
                                             &result_length,
                                             result_buffer,
                                             &result_length,
                                             &text_context->text_a_file_handle);
#ifdef vms
                /* Parse filename from file specification
                 * for use as the Title field in the Header
                 */
                if (SUCCESS(status))
                {
                        struct FAB fil_fab;     /* File access block */
                        struct NAM fil_nam;     /* Name block */
                        unsigned long   esa_length = 255   /* file length */
                        unsigned char   esa_buffer[255];   /* file buffer */

                        /* Initialize fab and nam blocks */
                        fil_fab = cc$rms_fab;
                        fil_nam = cc$rms_nam;
```

```
                        fil_fab.fab$l_dna = 0;
                        fil_fab.fab$b_dns = 0;
                        fil_fab.fab$l_fna = result_buffer;
                        fil_fab.fab$b_fns = result_length;
                        fil_fab.fab$l_nam = &fil_nam;
                        fil_fab.fab$l_fop = FAB$M_NAM;

                        fil_nam.nam$b_nop = NAM$M_SYNCHK;
                        fil_nam.nam$l_rlf = 0;
                        fil_nam.nam$l_esa = esa_buffer;
                        fil_nam.nam$b_ess = esa_length;

                        /* Parse the file specification */
                        status = sys$parse(&fil_fab);
                        if (FAILURE(status))
                                return (status);

                        /* Copy the filename into the title area */
                        text_context->text_l_title_length = fil_nam.nam$b_name;
                        strncpy(text_context->text_a_title,
                                fil_nam.nam$l_name,
                                fil_nam.nam$b_name);

                        /* Copy the file extension into the title area */
                        strncpy(text_context->text_a_title +
                                        text_context->text_l_title_length,
                                fil_nam.nam$l_type,
                                fil_nam.nam$b_type);
                        text_context->text_l_title_length += fil_nam.nam$b_type;
                }
#endif
        }

        /* If an input procedure was specified, set
         * the position parameter to the input parameter
         * otherwise, use the file handle.
         */
        if (text_context->text_a_input_routine != 0)
                text_context->text_a_position_param =
                        text_context->text_a_input_routine_param;
        else
                text_context->text_a_position_param =
                        text_context->text_a_file_handle;

        /*
         * The state value tells the Get Aggregate routine what
         * aggregate to return next. In this case (first), we want
         * it to return a document descriptor.
         */
        text_context->text_l_state = DDIF$_DSC;                          ❽

        /* Fill in get and close procedure addresses */
        *text_context_ptr = (unsigned long) text_context;
        *get_aggr         = (unsigned long) get_aggregate;
        *get_pos          = (unsigned long) get_position;
        *close_text       = (unsigned long) close_front_end;

        /* How did we do? */
        return status;
}
```

The following callouts correspond to the callouts in the *get_aggregate* routine in the Text front end.

❾ This routine reads the input data and calls the appropriate routines to create the necessary aggregates. Before doing so, however, this routine

creates a DDIF$_DSC aggregate and a DDIF$_DHD aggregate, both of which are required in every DDIF document.

❿ Before reading the input and creating the appropriate content aggregates, this routine creates a document descriptor (DDIF$_DSC) and document header (DDIF$_DHD) aggregate. These aggregates, along with the document root aggregate, are required in every DDIF document.

The **text_context->text_1_state** argument is used to specify the next aggregate to be created. After the DDIF$_DSC and DDIF$_DHD aggregates have been created, the state is set to DDIF$_SEG, so that the next aggregate created will be the root segment aggregate.

```
/*
**++
**  FUNCTIONAL DESCRIPTION:                                              ❾
**
**      This routine is the entry point for the 'get_aggregate' procedure.
**      It reads an aggregate from the input DDIF stream and returns
**      this aggregate to the caller.
**
**  FORMAL PARAMETERS:
**
**      text_context.wlu.v      value to identify this converter instance
**
**      aggregate_handle.wlu.r  address to store aggregate handle
**
**      aggregate_type.wlu.r    address to store aggregate type
**
**  IMPLICIT INPUTS:
**
**      none
**
**  IMPLICIT OUTPUTS:
**
**      none
**
**  FUNCTION VALUE:
**
**      CDA$_NORMAL
**      CDA$_ENDOFDOC
**      Memory allocation error conditions
**      File error conditions
**
**  SIDE EFFECTS:
**
**      none
**
**--
**/
static unsigned long   get_aggregate (text_context_ptr,
                                      aggregate_handle,
                                      aggregate_type)
unsigned long           *text_context_ptr;
unsigned long           *aggregate_handle;
unsigned long           *aggregate_type;

{
unsigned long   status;
struct text_cxt *text_context;
```

```
/* Dereference */
text_context = (struct text_cxt *) *text_context_ptr;

/*
 * The state value tells the Get Aggregate routine what aggregate
 * to return next. We will test the state value here to determine
 * what type of aggregate is needed. Each time an aggregate is
 * returned, the state value is set to return the next type of
 * aggregate.
 */
/* Find what DDIF aggregate we need to return */

switch (text_context->text_l_state)
{

    /* Build a document descriptor */
    case DDIF$_DSC:                                                    (10)

        status = create_dsc (&text_context,
                             aggregate_type,
                             aggregate_handle);
        break;

    /* Build a document header */
    case DDIF$_DHD:

        status = create_dhd (&text_context,
                             aggregate_type,
                             aggregate_handle);
        break;

    /* Build a document segment */
    case DDIF$_SEG:

        /* Create the SEG aggregate */
        status = create_seg (&text_context,
                             aggregate_type,
                             aggregate_handle);

        break;

    /* Build a text aggregate */
    case DDIF$_TXT:

        /* Create a TXT aggregate */
        status = create_txt (&text_context,
                             aggregate_type,
                             aggregate_handle);
        break;

    /* Build a directive (new_line or new_page) */
    case DDIF$_SFT:
    case DDIF$_HRD:

        /* Create a hard or soft directive aggregate */
        status = create_dir (&text_context,
                             aggregate_type,
                             aggregate_handle);
        break;

    /* Build an end of segment */
    case DDIF$_EOS:

        /* Create an end of segment aggregate */
        status = create_eos (&text_context,
                             aggregate_type,
                             aggregate_handle);
        break;
```

```
            /* If we got here it is surely an insidious bug */
            default:
                status = CDA$_INTERR;
                break;
        }

        /* Return the status */
        return status;
}
```

The following callout corresponds to the callout in the *create_dsc* routine in the Text front end.

❶ This routine creates and fills in the required DDIF$_DSC aggregate, sets the state to DDIF$_DHD, and returns to the switch statement referenced by ❿.

```
/*
**++
**  FUNCTIONAL DESCRIPTION:                                              ❶
**
**      This routine creates a document descriptor aggregate and
**      fills it in.
**
**  FORMAL PARAMETERS:
**
**      text_context.wlu.v              value to identify this converter
**
**      aggregate_type.wlu.r            pointer to aggregate type
**
**      aggregate_handle.wlu.r          pointer to aggregate handle
**
**  IMPLICIT INPUTS:
**
**      none
**
**  IMPLICIT OUTPUTS:
**
**      none
**
**  FUNCTION VALUE:
**
**      CDA$_NORMAL
**      Aggregate creation errors
**      Memory deallocation error conditions
**
**  SIDE EFFECTS:
**
**      none
**
**--
**/
static unsigned long   create_dsc (text_context_ptr,
                                   aggregate_type,
                                   aggregate_handle)

unsigned long           *text_context_ptr;
unsigned long           *aggregate_type;
unsigned long           *aggregate_handle;
```

```
{
unsigned long   status;
struct text_cxt *text_context;
unsigned long   aggregate_item;
unsigned long   item_length;
unsigned long   item_index = 0;
unsigned long   add_info;
unsigned long   major_version;
unsigned long   minor_version;

        /* Dereference */
        text_context = (struct text_cxt *) *text_context_ptr;

        /* Set the aggregate type */
        *aggregate_type = DDIF$_DSC;

        /* Create the aggregate */
        status = cda$create_aggregate
                        (&text_context->text_a_root_aggregate_handle,
                         aggregate_type,
                         aggregate_handle);
        if (FAILURE(status))
                return (status);

        /* First item to include is the major version. */
        major_version = DDIF$K_MAJOR_VERSION;
        item_length = sizeof(major_version);
        aggregate_item = DDIF$_DSC_MAJOR_VERSION ;
        status = cda$store_item (&text_context->text_a_root_aggregate_handle,
                                 aggregate_handle,
                                 &aggregate_item,
                                 &item_length,
                                 &major_version);
        if (FAILURE(status))
                return (status);

        /* The next item is the minor version */
        minor_version = DDIF$K_MINOR_VERSION;
        item_length = sizeof(minor_version);
        aggregate_item = DDIF$_DSC_MINOR_VERSION ;
        status = cda$store_item (&text_context->text_a_root_aggregate_handle,
                                 aggregate_handle,
                                 &aggregate_item,
                                 &item_length,
                                 &minor_version);
        if (FAILURE(status))
                return (status);

        /* Now the product identifier */
        aggregate_item = DDIF$_DSC_PRODUCT_IDENTIFIER;
        status = cda$store_item (&text_context->text_a_root_aggregate_handle,
                                 aggregate_handle,
                                 &aggregate_item,
                                 &dsc_id_length,
                                 dsc_identifier);
        if (FAILURE(status))
                return (status);
```

```
        /* And the product name */
        aggregate_item = DDIF$_DSC_PRODUCT_NAME ;
        add_info = CDA$K_ISO_LATIN1;
        status = cda$store_item (&text_context->text_a_root_aggregate_handle,
                                 aggregate_handle,
                                 &aggregate_item,
                                 &dsc_nam_length,
                                 dsc_prod_name,
                                 &item_index,
                                 &add_info);

        /* Document header next */
        text_context->text_l_state= DDIF$_DHD;

        /* Say how we did */
        return (status);
}
```

The following callout corresponds to the callout in the *create_dhd* routine in the Text front end.

⓬ This routine creates and fills in the required DDIF$_DHD aggregate, sets the state to DDIF$_SEG, and returns to the switch statement referenced by ⓾.

```
/*
**++
**  FUNCTIONAL DESCRIPTION:                                              ⓬
**
**      This routine creates a document header aggregate and
**      fills it in.
**
**  FORMAL PARAMETERS:
**
**      text_context.wlu.v              value to identify this converter
**
**      aggregate_type.wlu.r            pointer to aggregate type
**
**      aggregate_handle.wlu.r          pointer to aggregate handle
**
**  IMPLICIT INPUTS:
**
**      none
**
**  IMPLICIT OUTPUTS:
**
**      none
**
**  FUNCTION VALUE:
**
**      CDA$_NORMAL
**      Aggregate creation errors
**      Memory deallocation error conditions
**
**  SIDE EFFECTS:
**
**      none
**
**--
**/
static unsigned long    create_dhd (text_context_ptr,
                                    aggregate_type,
                                    aggregate_handle)
```

```
unsigned long           *text_context_ptr;
unsigned long           *aggregate_type;
unsigned long           *aggregate_handle;

{
unsigned long  status;          /* return status */
struct text_cxt *text_context;  /* points to context block */
unsigned long  aggregate_item;
unsigned long  item_index = 0;
unsigned long  int_length;
unsigned long  add_info;
unsigned long  erf_type;
unsigned long  erf_handle;
unsigned char  *erf_aggregate;
unsigned long  object_identifier[7];

        /* Dereference */
        text_context = (struct text_cxt *) *text_context_ptr;

        /* Set the aggregate type to document header */
        *aggregate_type = DDIF$_DHD;
        add_info = CDA$K_ISO_LATIN1;

        /* Create the aggregate */
        status = cda$create_aggregate
                        (&text_context->text_a_root_aggregate_handle,
                         aggregate_type,
                         aggregate_handle);
        if (FAILURE(status))
                return (status);

        /* Fill in the Author */
        aggregate_item = DDIF$_DHD_AUTHOR;
        status = cda$store_item (&text_context->text_a_root_aggregate_handle,
                                 aggregate_handle,
                                 &aggregate_item,
                                 &dhd_aut_length,
                                 dhd_author,
                                 &item_index,
                                 &add_info);

        /* Fill in the Title if we have one */
        if ((text_context->text_l_title_length != 0) &&
            (SUCCESS(status)))
        {
                aggregate_item = DDIF$_DHD_TITLE;
                status = cda$store_item
                               (&text_context->text_a_root_aggregate_handle,
                                aggregate_handle,
                                &aggregate_item,
                                &text_context->text_l_title_length,
                                text_context->text_a_title,
                                &item_index,
                                &add_info);
        }

        /* Create and external reference aggregate */
        erf_type = DDIF$_ERF;

        /* Create the aggregate */
        status = cda$create_aggregate
                        (&text_context->text_a_root_aggregate_handle,
                         &erf_type,
                         &erf_handle);
        if (FAILURE(status))
                return (status);
```

```
/* Store the object identifier of DDIF */
object_identifier[0] = 1;
object_identifier[1] = 3;
object_identifier[2] = 12;
object_identifier[3] = 1011;
object_identifier[4] = 1;
object_identifier[5] = 3;
object_identifier[6] = 1;
aggregate_item = DDIF$_ERF_DATA_TYPE;
int_length = sizeof(object_identifier);
status = cda$store_item (&text_context->text_a_root_aggregate_handle,
                         &erf_handle,
                         &aggregate_item,
                         &int_length,
                         object_identifier);
if (FAILURE(status))
        return (status);

/* Store the style guide name */
aggregate_item = DDIF$_ERF_LABEL;
add_info = CDA$K_ISO_LATIN1;
status = cda$store_item (&text_context->text_a_root_aggregate_handle,
                         &erf_handle,
                         &aggregate_item,
                         &style_length,
                         style_guide_name,
                         &item_index,
                         &add_info);
if (FAILURE(status))
        return (status);

/* Store the descriptor */
aggregate_item = DDIF$_ERF_DESCRIPTOR;
add_info = CDA$K_ISO_LATIN1;
item_index = 0;
status = cda$store_item (&text_context->text_a_root_aggregate_handle,
                         &erf_handle,
                         &aggregate_item,
                         &erf_desc_length,
                         erf_desc_type,
                         &item_index,
                         &add_info);
if (FAILURE(status))
        return (status);

/* Store the label type */
aggregate_item = DDIF$_ERF_LABEL_TYPE;
add_info = DDIF$K_STYLE_LABEL_TYPE;
status = cda$store_item (&text_context->text_a_root_aggregate_handle,
                         &erf_handle,
                         &aggregate_item,
                         &erf_length,
                         erf_label_type,
                         &item_index,
                         &add_info);
if (FAILURE(status))
        return (status);
```

```
        /* Store the copy info */
        aggregate_item = DDIF$_ERF_CONTROL;
        int_length = sizeof(unsigned long);
        item_index = DDIF$K_NO_COPY_REFERENCE;
        status = cda$store_item (&text_context->text_a_root_aggregate_handle,
                                 &erf_handle,
                                 &aggregate_item,
                                 &int_length,
                                 &item_index);
        if (FAILURE(status))
                return (status);

        /* Store the Style Guide External Reference */
        aggregate_item = DDIF$_DHD_EXTERNAL_REFERENCES;
        int_length = sizeof(unsigned long);
        status = cda$store_item (&text_context->text_a_root_aggregate_handle,
                                 aggregate_handle,
                                 &aggregate_item,
                                 &int_length,
                                 &erf_handle);
        if (FAILURE(status))
                return (status);

        /* Fill in the Style Guide */
        aggregate_item = DDIF$_DHD_STYLE_GUIDE;
        item_index = 1;
        int_length = sizeof(unsigned long);
        status = cda$store_item (&text_context->text_a_root_aggregate_handle,
                                 aggregate_handle,
                                 &aggregate_item,
                                 &int_length,
                                 &item_index);

        /* Segment next */
        text_context->text_l_state= DDIF$_SEG;

        /* Say how we did */
        return status;
}
```

The following callouts correspond to the callouts in the *create_seg* routine in the Text front end.

⓭ The first time this entry point is invoked, this routine creates the required document root segment and returns to the switch statement referenced by ⓾ with the state still set to DDIF$_SEG. All subsequent calls to this routine create nested segments that contain the document content.

⓮ If the root segment has just been created, this routine also creates a segment attributes aggregate (type DDIF$_SGA) and a type definition aggregate (type DDIF$_TYD) to define types that are accessible to all of the document content aggregates. Once these aggregates are created, this routine passes control back to the switch statement referenced by ⓾. Because the state is still set to DDIF$_SEG, ⓾ immediately passes control back to this routine to create the first nested segment of the document.

⓯ If this routine is not creating the root segment, it simply creates a nested segment aggregate and sets the state to DDIF$_TXT before passing control back to ⓾.

```
/*
**++
**  FUNCTIONAL DESCRIPTION:                                                    (13)
**
**      This routine creates a document segment aggregate and
**      fills it in.
**
**  FORMAL PARAMETERS:
**
**      text_context.wlu.v              value to identify this converter
**
**      aggregate_type.wlu.r            pointer to aggregate type
**
**      aggregate_handle.wlu.r          pointer to aggregate handle
**
**  IMPLICIT INPUTS:
**
**      none
**
**  IMPLICIT OUTPUTS:
**
**      none
**
**  FUNCTION VALUE:
**
**      CDA$_NORMAL
**      Aggregate creation errors
**      Memory deallocation error conditions
**
**  SIDE EFFECTS:
**
**      none
**
**--
**/
static unsigned long    create_seg (text_context_ptr,
                                    aggregate_type,
                                    aggregate_handle)

unsigned long           *text_context_ptr;
unsigned long           *aggregate_type;
unsigned long           *aggregate_handle;

{
unsigned long   status;
struct text_cxt *text_context;
unsigned long   aggregate_item;
unsigned long   item_length;
unsigned long   item_index = 0;
unsigned long   add_info;
unsigned long   tyd_handle;
unsigned long   tyd_type;
unsigned long   sga_handle;
unsigned long   sga_type;

        /* Dereference */
        text_context = (struct text_cxt *) *text_context_ptr;

        /* Set the aggregate type to segment */
        *aggregate_type = DDIF$_SEG;
```

```
/* Create the root segment */
status = cda$create_aggregate (&text_context->text_a_root_aggregate_handle,
                               aggregate_type,
                               aggregate_handle);
if (FAILURE(status))
        return (status);

/* If this is the root segment, then setup to create a  */
/* child segment.  */
if (text_context->text_v_root_segment == 1)                    (14)
{
        /* Reset flags */
        text_context->text_v_root_segment = 0;

        /* Store SEG ID in segment */
        aggregate_item = DDIF$_SEG_ID;
        status = cda$store_item
                        (&text_context->text_a_root_aggregate_handle,
                         aggregate_handle,
                         &aggregate_item,
                         &seg_id_length,
                         seg_id);
        if (FAILURE(status))
                return (status);

        /* Create an attribute aggregate */
        sga_type = DDIF$_SGA;
        status = cda$create_aggregate
                        (&text_context->text_a_root_aggregate_handle,
                         &sga_type,
                         &sga_handle);
        if (FAILURE(status))
                return (status);

        /* Store SGA in segment */
        aggregate_item = DDIF$_SEG_SPECIFIC_ATTRIBUTES;
        item_length = sizeof (sga_handle);
        status = cda$store_item
                        (&text_context->text_a_root_aggregate_handle,
                         aggregate_handle,
                         &aggregate_item,
                         &item_length,
                         &sga_handle);
        if (FAILURE(status))
                return (status);

        /* Create a type definition aggregate */
        tyd_type = DDIF$_TYD;
        status = cda$create_aggregate
                        (&text_context->text_a_root_aggregate_handle,
                         &tyd_type,
                         &tyd_handle);
        if (FAILURE(status))
                return (status);
```

```
                /* Store TYD in SGA */
                aggregate_item = DDIF$_SGA_TYPE_DEFNS;
                item_length = sizeof (tyd_handle);
                status = cda$store_item
                                (&text_context->text_a_root_aggregate_handle,
                                 &sga_handle,
                                 &aggregate_item,
                                 &item_length,
                                 &tyd_handle);
                if (FAILURE(status))
                        return (status);

                /* Store TYD_LABEL in TYD */
                aggregate_item = DDIF$_TYD_LABEL;
                status = cda$store_item
                                (&text_context->text_a_root_aggregate_handle,
                                 &tyd_handle,
                                 &aggregate_item,
                                 &para_length,
                                 para_buffer);
                if (FAILURE(status))
                        return (status);

                /* Store TYD_PARENT in TYD */
                aggregate_item = DDIF$_TYD_PARENT;
                status = cda$store_item
                                (&text_context->text_a_root_aggregate_handle,
                                 &tyd_handle,
                                 &aggregate_item,
                                 &literal_length,
                                 literal_buffer);
                if (FAILURE(status))
                        return (status);
        }
        else
        {
                /* Not a root segment; tag as paragraph */
                aggregate_item = DDIF$_SEG_SEGMENT_TYPE;                    ⓯
                status = cda$store_item
                                (&text_context->text_a_root_aggregate_handle,
                                 aggregate_handle,
                                 &aggregate_item,
                                 &para_length,
                                 para_buffer);
                if (FAILURE(status))
                        return (status);

                text_context->text_l_state= DDIF$_TXT;
        }

        /* Bump scope level */
        text_context->text_b_scope_level += 1;

        /* Say how we did */
        return status;
}
```

The following callouts correspond to the callouts in the *create_txt* routine in the Text front end.

⓰ This routine creates and fills in a text content aggregate.

⓱ If a user-supplied text file input procedure was specified in the item list, use that procedure. Otherwise, use the CDA Toolkit routine READ TEXT FILE.

⓲ If we reached the end of the document, pass control back to ⓾.

⓳ This loop reads each character on the line of text. If a form-feed character is encoutered, the **ff_found** flag is set.

⓴ If a horizontal tab character is encountered, the **ht_found** flag is set.

㉑ The characters are passed through a filter to ensure that there are no control characters.

㉒ If **write_length** was not zero, there was text on the line, so a DDIF$_TXT aggregate is created and the text is stored in the aggregate.

㉓ If a form-feed character was encountered (indicated by **ff_found** = 1), this corresponds to a DDIF hard directive. Therefore, the value of the directive is set to DDIF$K_DIR_NEW_PAGE and the state is set to DDIF$_HRD.

㉔ If a tab character was encountered (indicated by **ht_found** = 1), this corresponds to a DDIF soft directive. Therefore, the value of the directive is set to DDIF$K_DIR_TAB and the state is set to DDIF$_SFT.

㉕ If the tab or form-feed directive was the first character encountered on the line, pass control to the *create_dir* entry point to create the necessary directive aggregate.

㉖ If there was no form-feed or horizontal tab directive on the line, this statement checks to see if the line was completely read or if there are more characters on the line to be processed. If the line has been completely read, the next aggregate to be created is a new line (DDIF$K_DIR_NEW_LINE) soft directive aggregate (type DDIF$_SFT). Otherwise, create another DDIF$_TXT aggregate because there is more text to read.

㉗ If the line was empty, the next aggregate to be created is new line (DDIF$K_DIR_NEW_LINE) soft directive aggregate (type DDIF$_SFT). If this is the case, the value of the directive is set to DDIF$K_DIR_NEW_LINE, the state is set to DDIF$_SFT, and the *create_dir* routine is invoked.

```
/*
**++
**  FUNCTIONAL DESCRIPTION:                                              ⓰
**
**      This routine creates a text aggregate and fills it in.
**
**  FORMAL PARAMETERS:
**
**      text_context.wlu.v              value to identify this converter
**
**      aggregate_type.wlu.r            pointer to aggregate type
**
**      aggregate_handle.wlu.r          pointer to aggregate handle
**
**  IMPLICIT INPUTS:
**
**      none
**
```

```
**  IMPLICIT OUTPUTS:
**
**      none
**
**  FUNCTION VALUE:
**
**      CDA$_NORMAL
**      Aggregate creation errors
**      Memory deallocation error conditions
**
**  SIDE EFFECTS:
**
**      none
**
**--
**/
static unsigned long    create_txt (text_context_ptr,
                                    aggregate_type,
                                    aggregate_handle)

unsigned long           *text_context_ptr;
unsigned long           *aggregate_type;
unsigned long           *aggregate_handle;

{
unsigned long   status;
struct text_cxt *text_context;
unsigned long   aggregate_item;
unsigned long   item_index;
unsigned long   add_info;
unsigned long   write_length;
unsigned long   ff_found;
unsigned long   ht_found;
unsigned long   junk;

        /* Dereference */
        text_context = (struct text_cxt *) *text_context_ptr;
        write_length = 0;
        ff_found     = 0;
        ht_found     = 0;
        item_index   = 0;

        /* Do we need to get a line of text from the text file? */
        if (text_context->text_l_buffer_length == 0)
        {
                /* File or procedure? */
                if (text_context->text_a_input_routine == 0)          (17)
                {
                        status = cda$read_text_file
                                        (&text_context->text_a_file_handle,
                                         &text_context->text_l_buffer_length,
                                         &text_context->text_a_buffer_address);
                }
                else
                {
                        status = (*text_context->text_a_input_routine)
                                     (text_context->text_a_input_routine_param,
                                      &text_context->text_l_buffer_length,
                                      &text_context->text_a_buffer_address);
                }
```

```
                /* Check for ENDOFDOC.  If found, then
                   stack for later processing. */
                if (status == CDA$_ENDOFDOC)
                {
                        text_context->text_v_end_of_document = 1;       ⓲

                        /* Create an end of segment aggregate */
                        status = create_eos (&text_context,
                                             aggregate_type,
                                             aggregate_handle);
                        /* Get out of here; no further processing in TXT */
                        return status;
                }

                if (FAILURE(status))
                        return (status);
                else
                        text_context->text_l_newline_count += 1;
        }

        /* Allocate text buffer */
        if (text_context->text_l_local_length < text_context->text_l_buffer_length)
        {
            /* Deallocate old one first */
            if text_context->text_l_local_length > 0)
#ifdef vms
               lib$free_vm(&text_context->text_l_local_length,
                           &text_context->text_a_local_buffer, 0);
#else
               free(text_context->text_a_local_buffer);
#endif

            /* Allocate larger buffer */
            if (DDIF_BUFFER_SIZE > text_context->text_l_buffer_length)
                text_context->text_l_local_length = DDIF_BUFFER_SIZE;
            else
                text_context->text_l_local_length =
                text_context->text_l_buffer_length;
#ifdef vms
            status = lib$get_vm(&text_context->text_l_local_length,
                                &text_context->text_a_local_buffer, 0);
#else
            text_context->text_a_local_buffer = (unsigned char *)
                                malloc(text_context->text_l_local_length);
            (text_context->text_a_local_buffer == 0) ?
                         (status = CDA$_ALLOCFAIL) : (status = 1);
#endif
            if (FAILURE(status))
                return (status);
        }

        /* Were there characters on the line? */
        if(text_context->text_l_buffer_length != 0)
        {
            while (write_length < text_context->text_l_buffer_length)  ⓳
            {
            /* Look for the Form Feed character (12) which is translated to
             * a new_page soft directive
             */
                if (text_context->text_a_buffer_address[write_length]
                                    ==  FORM_FEED)
                {
                     ff_found = 1;
                     break;
                }
```

```
    else
         if (text_context->text_a_buffer_address[write_length]
                     == HORIZONTAL_TAB)                          ⓴
         {
                 ht_found = 1;
                 break;
         }
         else
         {
            /* Make sure no control characters
             * pass through */                                    ㉑
            text_context->text_a_local_buffer[write_length]
                          = lookup_buffer
               [text_context->text_a_buffer_address[write_length]];

               write_length += 1;
          }
}

    /* Is there anything to write?  May not be if
       FF is first on line */
    if (write_length != 0)                                       ㉒
    {
          /* There was text on the line so
              we set the aggregate type to text */
            *aggregate_type = DDIF$_TXT;

            status = cda$create_aggregate
                     (&text_context->text_a_root_aggregate_handle,
                      aggregate_type,
                      aggregate_handle);
            if (FAILURE(status))
                    return (status);

            /* We now store the text line as a text content item */
            aggregate_item = DDIF$_TXT_CONTENT;
            add_info = CDA$K_ISO_LATIN1;
            status = cda$store_item
                    (&text_context->text_a_root_aggregate_handle,
                     aggregate_handle,
                     &aggregate_item,
                     &write_length,
                     text_context->text_a_local_buffer,
                     &item_index,
                     &add_info);
            if (FAILURE(status))
                    return (status);

            /* Adjust buffer count and address for next pass */
            text_context->text_l_buffer_length  -= write_length;
            text_context->text_a_buffer_address += write_length;
    }

    /* Special case for FORM_FEED or HORIZONTAL_TAB characters;
       skip over it */
    if ((ff_found == 1) ||
        (ht_found == 1))
    {
            text_context->text_l_buffer_length  -= 1;
            text_context->text_a_buffer_address += 1;
```

```
                        /* Setup for directive */
                        if (ff_found == 1)                                    (23)
                        {
                                text_context->text_l_directive_content =
                                                DDIF$K_DIR_NEW_PAGE;
                                text_context->text_l_state =
                                                DDIF$_HRD;
                                text_context->text_l_directive_type =
                                                DDIF$_HRD;
                        }
                        else                                                  (24)
                        {
                                text_context->text_l_directive_content =
                                                DDIF$K_DIR_TAB;
                                text_context->text_l_state = DDIF$_SFT;
                                text_context->text_l_directive_type = DDIF$_SFT;
                        }

                        /* Create a directive aggregate if it is
                           first on line */                                   (25)
                        if (write_length == 0)
                        {
                                status = create_dir (&text_context,
                                                     aggregate_type,
                                                     aggregate_handle);
                        }
                }

                /* Finished with the line? */                                 (26)
                else
                        if (text_context->text_l_buffer_length == 0)
                        {
                                /* Set next aggregate as new_line directive */
                                text_context->text_l_directive_content =
                                                DDIF$K_DIR_NEW_LINE;
                                text_context->text_l_state = DDIF$_SFT;
                                text_context->text_l_directive_type = DDIF$_SFT;
                        }
                        else
                                /* Otherwise, next aggregate is TXT */
                                text_context->text_l_state= DDIF$_TXT;
        }

        /* Empty line */                                                      (27)
        else
        {
                /* Set directive to be new line */
                text_context->text_l_directive_content = DDIF$K_DIR_NEW_LINE;
                text_context->text_l_directive_type    = DDIF$_SFT;

                /* Create a directive aggregate */
                status = create_dir (&text_context,
                                     aggregate_type,
                                     aggregate_handle);
        }

        /* Say how we did */
        return status;
}
```

The following callouts correspond to the callouts in the *create_eos* routine in the Text front end.

(28) This routine creates an end-of-segment (type DDIF$_EOS) aggregate. This aggregate is a "dummy" aggregate in that it is not actually stored

in the DDIF document. Instead, it is used to indicate the end of a segment.

29 If the front end has reached the end of the document and if the scope level is greater than or equal to 1 (the scope level indicates the level of nesting of segments), the previous DDIF$_EOS aggregate completed a nested segment and there are more segments to be completed before the document itself can be completed. In this case, the routine must continue to create DDIF$_EOS aggregates until the scope level is 0, meaning that the end of the root segment has been reached. At that point, the status CDA$_ENDOFDOC can be returned.

30 If the front end has not reached the end of the document, this routine only creates one DDIF$_EOS aggregate to complete the current nested segment. In this case, the state is set to DDIF$_SEG so that the next aggregate created is another nested segment.

31 This statement decrements the scope level to indicate that a nested segment has been completed by a DDIF$_EOS aggregate.

```
/*
**++
**  FUNCTIONAL DESCRIPTION:                                              28
**
**      This routine creates an end of segment aggregate
**
**  FORMAL PARAMETERS:
**
**      text_context.wlu.v              value to identify this converter
**
**      aggregate_type.wlu.r            pointer to aggregate type
**
**      aggregate_handle.wlu.r          pointer to aggregate handle
**
**  IMPLICIT INPUTS:
**
**      none
**
**  IMPLICIT OUTPUTS:
**
**      none
**
**  FUNCTION VALUE:
**
**      CDA$_NORMAL
**      Aggregate creation errors
**      Memory deallocation error conditions
**
**  SIDE EFFECTS:
**
**      none
**
**--
**/
static unsigned long    create_eos (text_context_ptr,
                                    aggregate_type,
                                    aggregate_handle)

unsigned long           *text_context_ptr;
unsigned long           *aggregate_type;
unsigned long           *aggregate_handle;
```

```
{
unsigned long   status;
struct text_cxt *text_context;

        /* Dereference */
        text_context = (struct text_cxt *) *text_context_ptr;

        /* Return EOS as current aggregate */
        *aggregate_type   = DDIF$_EOS;
        *aggregate_handle = 0;

        /* If end of document, then set status */
        if (text_context->text_v_end_of_document == 1)
        {
                if (text_context->text_b_scope_level >= 1)              (29)
                {
                        /* Set next directive to be EOS for content */
                        text_context->text_l_state= DDIF$_EOS;

                        /* Set status to success */
                        status = CDA$_NORMAL;
                }
                else
                        /* Set status to end of document */
                        status = CDA$_ENDOFDOC;
        }
        else
        {
                /* Set state to be SEG*/                                (30)
                text_context->text_l_state= DDIF$_SEG;

                /* Set status to success */
                status = CDA$_NORMAL;
        }

        /* Decrement scope level */
        text_context->text_b_scope_level -= 1;                          (31)

        return (status);
}
```

The following callout corresponds to the callout in the *look_ahead* routine in the Text front end.

(32) This routine is called by the *create_dir* routine to scan through multiple blank lines in the text file.

```
/*
**++
**  FUNCTIONAL DESCRIPTION:                                             (32)
**
**      This routine looks ahead for multiple blank lines in the text stream.
**      Multiple blank lines indicate end of paragraph.  They become
**      hard newline directives.
**
**  FORMAL PARAMETERS:
**
**      text_context.wlu.v              value to identify this converter
**
**      aggregate_type.wlu.r            pointer to aggregate type
**
**      aggregate_handle.wlu.r          pointer to aggregate handle
**
```

```
**  IMPLICIT INPUTS:
**
**      none
**
**  IMPLICIT OUTPUTS:
**
**      none
**
**  FUNCTION VALUE:
**
**      CDA$_NORMAL
**      Aggregate creation errors
**      Memory deallocation error conditions
**
**  SIDE EFFECTS:
**
**      none
**
**--
**/
static unsigned long    look_ahead (text_context_ptr)

unsigned long           *text_context_ptr;

{
unsigned long   status = 1;
struct text_cxt *text_context;

        /* Dereference */
        text_context = (struct text_cxt *) *text_context_ptr;

        /* Look ahead and compress blank lines */
        while ((text_context->text_l_buffer_length == 0) &
                (SUCCESS(status)))
        {
                /* File or procedure? */
                if (text_context->text_a_input_routine == 0)
                {
                        status = cda$read_text_file
                                        (&text_context->text_a_file_handle,
                                        &text_context->text_l_buffer_length,
                                        &text_context->text_a_buffer_address);
                }
                else
                {
                        status = (*text_context->text_a_input_routine)
                                 (text_context->text_a_input_routine_param,
                                  &text_context->text_l_buffer_length,
                                  &text_context->text_a_buffer_address);
                 }
                if (SUCCESS(status))
                        text_context->text_l_newline_count += 1;
        }

        /* Check for ENDOFDOC.  If found, then stack for later processing. */
        if (status == CDA$_ENDOFDOC)
        {
                text_context->text_v_end_of_document = 1;
                status = CDA$_NORMAL;
        }

        return status;
}
```

The following callouts correspond to the callouts in the *create_dir* routine in the Text front end.

33 If the directive content was set to DDIF$K_DIR_NEW_LINE (regardless of whether it indicates the end of a paragraph or the end of the document), this directive must be stored as a hard directive in a DDIF$_HRD aggregate.

34 Otherwise, the appropriate type of aggregate is created and filled in.

35 If the directive was a new-line directive, the new-line counter is decremented and the routine checks to see if it is at the end of a paragraph, the end of the document, or if there are more new lines to process. The appropriate values are specified according to which case applies.

```
/*
**++
**  FUNCTIONAL DESCRIPTION:
**
**      This routine creates a directive aggregate and
**      fills it in.
**
**  FORMAL PARAMETERS:
**
**      text_context.wlu.v              value to identify this converter
**
**      aggregate_type.wlu.r            pointer to aggregate type
**
**      aggregate_handle.wlu.r          pointer to aggregate handle
**
**  IMPLICIT INPUTS:
**
**      none
**
**  IMPLICIT OUTPUTS:
**
**      none
**
**  FUNCTION VALUE:
**
**      CDA$_NORMAL
**      Aggregate creation errors
**      Memory deallocation error conditions
**
**  SIDE EFFECTS:
**
**      none
**
**--
**/
static unsigned long    create_dir (text_context_ptr,
                                    aggregate_type,
                                    aggregate_handle)

unsigned long           *text_context_ptr;
unsigned long           *aggregate_type;
unsigned long           *aggregate_handle;
```

```
{
unsigned long   status;
struct text_cxt *text_context;
unsigned long   aggregate_item;
unsigned long   item_length;

        /* Dereference */
        text_context = (struct text_cxt *) *text_context_ptr;

        /* Look ahead for blank lines? */
        if ((text_context->text_l_newline_count == 1) &&
            (text_context->text_v_end_of_paragraph == 0) &&
            (text_context->text_l_buffer_length == 0))
        {
                status = look_ahead (&text_context);
                if (FAILURE(status))
                        return (status);
        }

        /* Is this a new line? */                                          (33)
        if (text_context->text_l_directive_content == DDIF$K_DIR_NEW_LINE)
        {
                /* End of paragraph? (current newline plus at least 2 more) */
                if (text_context->text_l_newline_count > 2)
                        text_context->text_v_end_of_paragraph = 1;

                /* Set HRD directive if end of paragraph or document */
                if (text_context->text_v_end_of_paragraph == 1)
                        text_context->text_l_directive_type = DDIF$_HRD;

                if ((text_context->text_v_end_of_document == 1) &&
                    (text_context->text_l_newline_count == 1))
                     text_context->text_l_directive_type = DDIF$_HRD;
        }

        /* We are to return a directive */
        *aggregate_type = text_context->text_l_directive_type;

        /* Create the aggregate */                                         (34)
        status = cda$create_aggregate
                       (&text_context->text_a_root_aggregate_handle,
                        aggregate_type,
                        aggregate_handle);
        if (FAILURE(status))
                return (status);

        /* Set the directive type */
        if (text_context->text_l_directive_type == DDIF$_SFT)
                aggregate_item = DDIF$_SFT_DIRECTIVE;
        else
                aggregate_item = DDIF$_HRD_DIRECTIVE;

        /* Store it */
        item_length = sizeof(text_context->text_l_directive_content);
        status = cda$store_item (&text_context->text_a_root_aggregate_handle,
                                 aggregate_handle,
                                 &aggregate_item,
                                 &item_length,
                                 &text_context->text_l_directive_content);
        if (FAILURE(status))
                return (status);

        /* If this is a new line directive, then decrement counter */  (35)
        if (text_context->text_l_directive_content == DDIF$K_DIR_NEW_LINE)
                text_context->text_l_newline_count -= 1;
```

```
        /* Decide what aggregate to process next */
        /* End of Document? */
        if (text_context->text_v_end_of_document == 1)
        {
            /* Soft newlines to end of document */
            if (text_context->text_l_newline_count >= 1)
            {
                text_context->text_l_state = DDIF$_HRD;
                text_context->text_l_directive_type = DDIF$_HRD;
                text_context->text_l_directive_content = DDIF$K_DIR_NEW_LINE;
            }
            else
                /* EOS terminates paragraph and document */
                text_context->text_l_state= DDIF$_EOS;
        }
        else
            /* End of Paragraph? */
            if (text_context->text_v_end_of_paragraph == 1)
            {
                /* Hard newlines to end of paragraph */
                if (text_context->text_l_newline_count >= 2)
                {
                    text_context->text_l_state = DDIF$_HRD;
                    text_context->text_l_directive_type = DDIF$_HRD;
                    text_context->text_l_directive_content = DDIF$K_DIR_NEW_LINE;
                }
                else
                /* EOS terminates paragraph */
                {
                    text_context->text_l_state= DDIF$_EOS;
                    text_context->text_v_end_of_paragraph = 0;
                }
            }
            else
                /* Not end of paragraph or document, but more newlines */
                if (text_context->text_l_newline_count > 1)
                {
                    text_context->text_l_state= DDIF$_SFT;
                    text_context->text_l_directive_type = DDIF$_SFT;
                    text_context->text_l_directive_content = DDIF$K_DIR_NEW_LINE;
                }
                /* No more newlines; just text */
                else
                    text_context->text_l_state= DDIF$_TXT;

        /* Say how we did */
        return status;
}
```

The following callout corresponds to the callout in the *get-position* routine in the Text front end.

(36) This routine determines the current location of the front end within the input stream. This routine is used primarily by viewer applications for scroll bar support.

```
/*
**++
**  FUNCTIONAL DESCRIPTION:                                              (36)
**
**      This routine is the entry point for the 'get_position' procedure.
**      It returns the total size of the text stream and the current
**      position (or offset) within the text stream.
**
**  FORMAL PARAMETERS:
**
**      text_context.wlu.v      value to identify this converter instance
**
**      stream_position.wlu.r   address to store stream position
**
**      stream_size.wlu.r       address to store stream size
**
**  IMPLICIT INPUTS:
**
**      none
**
**  IMPLICIT OUTPUTS:
**
**      none
**
**  FUNCTION VALUE:
**
**      CDA$_NORMAL
**      CDA$_ENDOFDOC
**      Memory allocation error conditions
**      File error conditions
**
**  SIDE EFFECTS:
**
**      none
**
**--
**/
static unsigned long    get_position (text_context_ptr,
                                      stream_position,
                                      stream_size)
unsigned long           *text_context_ptr;
unsigned long           stream_position;
unsigned long           stream_size;

{
unsigned long   status;
struct text_cxt *text_context;

        /* Dereference */
        text_context = (struct text_cxt *) *text_context_ptr;

        /* Do we have a user supplied position routine? */
        if (text_context->text_a_position_routine == 0)
                /* Ask the CDA Toolkit for the position and size information */
                status = cda$get_text_position (&text_context->text_a_file_handle,
                                                stream_position,
                                                stream_size);
        else
                /* Ask user routine for position and size information */
                status = (*text_context->text_a_position_routine)
                                        (text_context->text_a_position_param,
                                        stream_position,
                                        stream_size);
```

```
        return status;
}
```

The following callout corresponds to the callout in the *close* routine in the Text front end.

(37) This routine closes the front end and deallocates all resources.

```
/*
**++
**  FUNCTIONAL DESCRIPTION:                                              (37)
**
**      This routine is the entry point for the 'close front end' procedure.
**      It closes the input DDIF file (or stream) and deallocates the
**      converter context.
**
**  FORMAL PARAMETERS:
**
**      text_context.wlu.v            value to identify this converter
**
**  IMPLICIT INPUTS:
**
**      none
**
**  IMPLICIT OUTPUTS:
**
**      none
**
**  FUNCTION VALUE:
**
**      CDA$_NORMAL
**      Memory deallocation error conditions
**      File error conditions
**
**  SIDE EFFECTS:
**
**      none
**
**--
**/
static unsigned long    close_front_end (text_context_ptr)

unsigned long    *text_context_ptr;

{
unsigned long   status;         /* return status */
unsigned long   struct_size;    /* holds context block size */
struct text_cxt *text_context;  /* points to context block */

        /* Dereference */
        text_context = (struct text_cxt *) *text_context_ptr;

        /* Do we have a file or just a stream? */
        status = CDA$_NORMAL;
        if (text_context->text_a_file_handle != 0)
        {
                /* Close the input file */
                status = cda$close_text_file
                                (&text_context->text_a_file_handle);
                if (FAILURE(status))
                        return (status);
        }
```

```
        /* Delete the root aggregate */
        status = cda$delete_root_aggregate
                                (&text_context->text_a_root_aggregate_handle);

        /* Deallocate text buffer and front end context block if we have one */
        struct_size = sizeof (struct text_cxt);
#ifdef vms
        if (text_context->text_l_local_length > 0)
            lib$free_vm(&text_context->text_l_local_length,
                        &text_context->text_a_local_buffer, 0);
        lib$free_vm (&struct_size, &text_context, 0);
#else
        if (text_context->text_l_local_length > 0)
            free(text_context->text_a_local_buffer);
        free(text_context);
#endif

        /* Say how we did */
        return status; }
```

D DDIF Aggregate Structures

This appendix lists the tables describing the structure and encoding of each DDIF aggregate.

Table D–1 lists the items in the document root aggregate and their encodings.

Table D–1 Document Root Aggregate (DDIF$_DDF)

Item Name	Item Encoding
DDIF$_DDF_DESCRIPTOR	Handle of DDIF$_DSC aggregate
DDIF$_DDF_HEADER	Handle of DDIF$_DHD aggregate
DDIF$_DDF_CONTENT	Handle of DDIF$_SEG aggregate

Table D–2 lists the items in the document descriptor aggregate and their encodings.

Table D–2 Document Descriptor Aggregate (DDIF$_DSC)

Item Name	Item Encoding
DDIF$_DSC_MAJOR_VERSION	Integer
DDIF$_DSC_MINOR_VERSION	Integer
DDIF$_DSC_PRODUCT_IDENTIFIER	String
DDIF$_DSC_PRODUCT_NAME	Array of type character string

Table D–3 lists the items in the document header aggregate and their encodings.

Table D–3 Document Header Aggregate (DDIF$_DHD)

Item Name	Item Encoding
DDIF$_DHD_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates
DDIF$_DHD_TITLE	Array of type character string
DDIF$_DHD_AUTHOR	Array of type character string
DDIF$_DHD_VERSION	Array of type character string
DDIF$_DHD_DATE	String
DDIF$_DHD_CONFORMANCE_TAGS	Array of type string with *add-info*

(continued on next page)

Table D–3 (Cont.) Document Header Aggregate (DDIF$_DHD)

Item Name	Item Encoding
DDIF$_DHD_EXTERNAL_REFERENCES	Sequence of DDIF$_ERF aggregates
DDIF$_DHD_LANGUAGES_C	Array of type enumeration
DDIF$_DHD_LANGUAGES	Array of type variable
DDIF$_DHD_STYLE_GUIDE	Integer

Table D–4 lists the items in the document segment aggregate and their encodings.

Table D–4 Document Segment Aggregate (DDIF$_SEG)

Item Name	Item Encoding
DDIF$_SEG_ID	String
DDIF$_SEG_USER_LABEL	Array of type character string
DDIF$_SEG_SEGMENT_TYPE	String
DDIF$_SEG_SPECIFIC_ATTRIBUTES	Handle of DDIF$_SGA aggregate
DDIF$_SEG_GENERIC_LAYOUT	Handle of DDIF$_LG1 aggregate
DDIF$_SEG_SPECIFIC_LAYOUT	Handle of DDIF$_LS1 aggregate
DDIF$_SEG_CONTENT	Sequence of content

Table D–5 lists the item in the Latin1 text content aggregate and its encoding.

Table D–5 Latin1 Text Content Aggregate (DDIF$_TXT)

Item Name	Item Encoding
DDIF$_TXT_CONTENT	String

Table D–6 lists the item in the general text content aggregate and its encoding.

Table D–6 General Text Content Aggregate (DDIF$_GTX)

Item Name	Item Encoding
DDIF$_GTX_CONTENT	Character string

Table D–7 lists the item in the hard directive aggregate and its encoding.

Table D–7 Hard Directive Aggregate (DDIF$_HRD)

Item Name	Item Encoding
DDIF$_HRD_DIRECTIVE	Enumeration

Table D–8 lists the item in the soft directive aggregate and its encoding.

Table D–8 Soft Directive Aggregate (DDIF$_SFT)

Item Name	Item Encoding
DDIF$_SFT_DIRECTIVE	Enumeration

Table D–9 lists the items in the hard value directive aggregate and their encodings.

Table D–9 Hard Value Directive Aggregate (DDIF$_HRV)

Item Name	Item Encoding
DDIF$_HRV_C	Enumeration
DDIF$_HRV_ESC_RATIO_N	Integer
DDIF$_HRV_ESC_RATIO_D	Integer
DDIF$_HRV_ESC_CONSTANT_C	Measurement enumeration
DDIF$_HRV_ESC_CONSTANT	Variable
DDIF$_HRV_RESET_VARIABLE	String
DDIF$_HRV_RESET_VALUE_C	Expression enumeration
DDIF$_HRV_RESET_VALUE	Variable

Table D–10 lists the items in the soft value directive aggregate and their encodings.

Table D–10 Soft Value Directive Aggregate (DDIF$_SFV)

Item Name	Item Encoding
DDIF$_SFV_C	Enumeration
DDIF$_SFV_ESC_RATIO_N	Integer
DDIF$_SFV_ESC_RATIO_D	Integer
DDIF$_SFV_ESC_CONSTANT_C	Measurement enumeration
DDIF$_SFV_ESC_CONSTANT	Variable
DDIF$_SFV_RESET_VARIABLE	String
DDIF$_SFV_RESET_VALUE_C	Expression enumeration
DDIF$_SFV_RESET_VALUE	Variable

Table D–11 lists the items in the Bézier curve aggregate and their encodings.

Table D–11 Bézier Curve Aggregate (DDIF$_BEZ)

Item Name	Item Encoding
DDIF$_BEZ_FLAGS	Longword
DDIF$_BEZ_PATH_C	Array of type measurement enumeration
DDIF$_BEZ_PATH	Array of type variable

Table D–12 lists the items in the polyline aggregate and their encodings.

Table D–12 Polyline Aggregate (DDIF$_LIN)

Item Name	Item Encoding
DDIF$_LIN_FLAGS	Longword
DDIF$_LIN_DRAW_PATTERN	Bit string
DDIF$_LIN_PATH_C	Array of type measurement enumeration
DDIF$_LIN_PATH	Array of type variable

Table D–13 lists the items in the arc content aggregate and their encodings.

Table D–13 Arc Content Aggregate (DDIF$_ARC)

Item Name	Item Encoding
DDIF$_ARC_FLAGS	Longword
DDIF$_ARC_CENTER_X_C	Measurement enumeration
DDIF$_ARC_CENTER_X	Variable
DDIF$_ARC_CENTER_Y_C	Measurement enumeration
DDIF$_ARC_CENTER_Y	Variable
DDIF$_ARC_RADIUS_X_C	Measurement enumeration
DDIF$_ARC_RADIUS_X	Variable
DDIF$_ARC_RADIUS_DELTA_Y_C	Measurement enumeration
DDIF$_ARC_RADIUS_DELTA_Y	Variable
DDIF$_ARC_START_C	AngleRef enumeration
DDIF$_ARC_START	Variable
DDIF$_ARC_EXTENT_C	AngleRef enumeration
DDIF$_ARC_EXTENT	Variable
DDIF$_ARC_ROTATION_C	AngleRef enumeration
DDIF$_ARC_ROTATION	Variable

Table D–14 lists the items in the fill area set content aggregate and their encodings.

Table D–14 Fill Area Set Content Aggregate (DDIF$_FAS)

Item Name	Item Encoding
DDIF$_FAS_FLAGS	Longword
DDIF$_FAS_PATH	Sequence of DDIF$_PTH aggregates

Table D–15 lists the items in the image content aggregate and their encodings.

Table D–15 Image Content Aggregate (DDIF$_IMG)

Item Name	Item Encoding
DDIF$_IMG_CONTENT	Sequence of DDIF$_IDU aggregates

Table D–16 lists the items in the content reference aggregate and their encodings.

Table D–16 Content Reference Aggregate (DDIF$_CRF)

Item Name	Item Encoding
DDIF$_CRF_TRANSFORM	Sequence of DDIF$_TRN aggregates
DDIF$_CRF_REFERENCE	String

Table D–17 lists the items in the external content aggregate and their encodings.

Table D–17 External Content Aggregate (DDIF$_EXT)

Item Name	Item Encoding
DDIF$_EXT_DIRECT_REFERENCE	Object identifier
DDIF$_EXT_INDIRECT_REFERENCE	Integer
DDIF$_EXT_DATA_VALUE_DESCRIPTOR	String
DDIF$_EXT_ENCODING_C	Enumeration
DDIF$_EXT_ENCODING	Variable
DDIF$_EXT_ENCODING_L	Integer

Table D–18 lists the items in the private content aggregate and their encodings.

Table D–18 Private Content Aggregate (DDIF$_PVT)

Item Name	Item Encoding
DDIF$_PVT_NAME	String
DDIF$_PVT_DATA_C	Enumeration
DDIF$_PVT_DATA	Variable
DDIF$_PVT_REFERENCE_ERF_INDEX	Integer

Table D–19 lists the items in the layout galley aggregate and their encodings.

Table D–19 Layout Galley Aggregate (DDIF$_GLY)

Item Name	Item Encoding
DDIF$_GLY_ID	String
DDIF$_GLY_BOUNDING_BOX_LL_X_C	Measurement enumeration
DDIF$_GLY_BOUNDING_BOX_LL_X	Variable
DDIF$_GLY_BOUNDING_BOX_LL_Y_C	Measurement enumeration
DDIF$_GLY_BOUNDING_BOX_LL_Y	Variable
DDIF$_GLY_BOUNDING_BOX_UR_X_C	Measurement enumeration
DDIF$_GLY_BOUNDING_BOX_UR_X	Variable
DDIF$_GLY_BOUNDING_BOX_UR_Y_C	Measurement enumeration
DDIF$_GLY_BOUNDING_BOX_UR_Y	Variable
DDIF$_GLY_OUTLINE	Sequence of DDIF$_PTH aggregates
DDIF$_GLY_FLAGS	Longword
DDIF$_GLY_STREAMS	Array of type string
DDIF$_GLY_SUCCESSOR_C	Enumeration
DDIF$_GLY_SUCCESSOR	Variable

Table D–20 lists the items in the external reference aggregate and their encodings.

Table D–20 External Reference Aggregate (DDIF$_ERF)

Item Name	Item Encoding
DDIF$_ERF_DATA_TYPE	Object identifier
DDIF$_ERF_DESCRIPTOR	Array of type character string
DDIF$_ERF_LABEL	Character string
DDIF$_ERF_LABEL_TYPE	String with *add-info*
DDIF$_ERF_CONTROL	Enumeration

Table D–21 lists the items in the image data unit aggregate and their encodings.

Table D–21 Image Data Unit Aggregate (DDIF$_IDU)

Item Name	Item Encoding
DDIF$_IDU_PRIVATE_CODING_ATTR	Sequence of DDIF$_PVT aggregates
DDIF$_IDU_PIXELS_PER_LINE	Integer
DDIF$_IDU_NUMBER_OF_LINES	Integer
DDIF$_IDU_COMPRESSION_TYPE	Enumeration
DDIF$_IDU_COMPRESSION_PARAMS	Sequence of DDIF$_PVT aggregates
DDIF$_IDU_DATA_OFFSET	Integer
DDIF$_IDU_PIXEL_STRIDE	Integer
DDIF$_IDU_SCANLINE_STRIDE	Integer
DDIF$_IDU_PIXEL_ORDER	Enumeration
DDIF$_IDU_BITS_PER_PIXEL	Integer
DDIF$_IDU_PLANE_DATA	String

Table D–22 lists the items in the composite path aggregate and their encodings.

Table D–22 Composite Path Aggregate (DDIF$_PTH)

Item Name	Item Encoding
DDIF$_PTH_C	Enumeration
DDIF$_PTH_LIN_PATH_C	Array of type measurement enumeration
DDIF$_PTH_LIN_PATH	Array of type variable
DDIF$_PTH_BEZ_PATH_C	Array of type measurement enumeration
DDIF$_PTH_BEZ_PATH	Array of type variable
DDIF$_PTH_ARC_CENTER_X_C	Measurement enumeration
DDIF$_PTH_ARC_CENTER_X	Variable

(continued on next page)

Table D–22 (Cont.) Composite Path Aggregate (DDIF$_PTH)

Item Name	Item Encoding
DDIF$_PTH_ARC_CENTER_Y_C	Measurement enumeration
DDIF$_PTH_ARC_CENTER_Y	Variable
DDIF$_PTH_ARC_RADIUS_X_C	Measurement enumeration
DDIF$_PTH_ARC_RADIUS_X	Variable
DDIF$_PTH_ARC_RADIUS_DELTA_Y_C	Measurement enumeration
DDIF$_PTH_ARC_RADIUS_DELTA_Y	Variable
DDIF$_PTH_ARC_START_C	AngleRef enumeration
DDIF$_PTH_ARC_START	Variable
DDIF$_PTH_ARC_EXTENT_C	AngleRef enumeration
DDIF$_PTH_ARC_EXTENT	Variable
DDIF$_PTH_ARC_ROTATION_C	AngleRef enumeration
DDIF$_PTH_ARC_ROTATION	Variable
DDIF$_PTH_REFERENCE	Integer

Table D–23 lists the items in the segment attributes aggregate and their encodings.

Table D–23 Segment Attributes Aggregate (DDIF$_SGA)

Item Name	Item Encoding
DDIF$_SGA_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates
DDIF$_SGA_CONTENT_STREAMS	Array of type string
DDIF$_SGA_CONTENT_CATEGORY	String with *add-info*
DDIF$_SGA_SEGMENT_TAGS	Array of type string with *add-info*
DDIF$_SGA_BINDING_DEFNS	Sequence of DDIF$_SGB aggregates
DDIF$_SGA_COMPUTE_C	Enumeration
DDIF$_SGA_CPTCPY_TARGET	String
DDIF$_SGA_CPTCPY_ERF_INDEX	Integer
DDIF$_SGA_CPTVAR_VARIABLE	String
DDIF$_SGA_CPTXRF_TARGET	String
DDIF$_SGA_CPTXRF_ERF_INDEX	Integer
DDIF$_SGA_CPTXRF_VARIABLE	String
DDIF$_SGA_CPTFNC_NAME	String
DDIF$_SGA_CPTFNC_PARAMETERS	Sequence of DDIF$_PVT aggregates
DDIF$_SGA_STRUCTURE_DESC_C	Enumeration
DDIF$_SGA_STRUCTURE_DESC	Sequence of DDIF$_OCC aggregates
DDIF$_SGA_LANGUAGE	Integer

(continued on next page)

Table D–23 (Cont.) Segment Attributes Aggregate (DDIF$_SGA)

Item Name	Item Encoding
DDIF$_SGA_LEGEND_UNIT_N	Integer
DDIF$_SGA_LEGEND_UNIT_D	Integer
DDIF$_SGA_LEGEND_UNIT_NAME	Array of type character string
DDIF$_SGA_UNITS_PER_MEASURE	Integer
DDIF$_SGA_UNITS_NAME	Array of type character string
DDIF$_SGA_ALT_PRESENTATION	Array of type character string
DDIF$_SGA_LAYOUT_C	Enumeration
DDIF$_SGA_LAYGLY_WRAP	Handle of DDIF$_LW1 aggregate
DDIF$_SGA_LAYGLY_LAYOUT	Handle of DDIF$_LL1 aggregate
DDIF$_SGA_LAYPTH_PATH	Sequence of DDIF$_PTH aggregates
DDIF$_SGA_LAYPTH_FORMAT	Enumeration
DDIF$_SGA_LAYPTH_ORIENTATION_C	Enumeration
DDIF$_SGA_LAYPTH_ORIENTATION	Variable
DDIF$_SGA_LAYPTH_H_ALIGN	Enumeration
DDIF$_SGA_LAYPTH_V_ALIGN	Enumeration
DDIF$_SGA_LAYREL_H_RATIO_N	Integer
DDIF$_SGA_LAYREL_H_RATIO_D	Integer
DDIF$_SGA_LAYREL_H_CONSTANT_C	Measurement enumeration
DDIF$_SGA_LAYREL_H_CONSTANT	Variable
DDIF$_SGA_LAYREL_V_RATIO_N	Integer
DDIF$_SGA_LAYREL_V_RATIO_D	Integer
DDIF$_SGA_LAYREL_V_CONSTANT_C	Measurement enumeration
DDIF$_SGA_LAYREL_V_CONSTANT	Variable
DDIF$_SGA_LAYPOS_TEXT_POSITION	Enumeration
DDIF$_SGA_FONT_DEFNS	Sequence of DDIF$_FTD aggregates
DDIF$_SGA_PATTERN_DEFNS	Sequence of DDIF$_PTD aggregates
DDIF$_SGA_PATH_DEFNS	Sequence of DDIF$_PHD aggregates
DDIF$_SGA_LINE_STYLE_DEFNS	Sequence of DDIF$_LSD aggregates
DDIF$_SGA_CONTENT_DEFNS	Sequence of DDIF$_CTD aggregates
DDIF$_SGA_TYPE_DEFNS	Sequence of DDIF$_TYD aggregates
DDIF$_SGA_TXT_MASK_PATTERN	Integer
DDIF$_SGA_TXT_FONT	Integer
DDIF$_SGA_TXT_RENDITION	Array of type enumeration
DDIF$_SGA_TXT_HEIGHT_C	Measurement enumeration
DDIF$_SGA_TXT_HEIGHT	Variable
DDIF$_SGA_TXT_SET_SIZE_N	Integer

(continued on next page)

Table D–23 (Cont.) Segment Attributes Aggregate (DDIF$_SGA)

Item Name	Item Encoding
DDIF$_SGA_TXT_SET_SIZE_D	Integer
DDIF$_SGA_TXT_DIRECTION	Enumeration
DDIF$_SGA_TXT_DEC_ALIGNMENT	Array of type character string
DDIF$_SGA_TXT_LEADER_SPACE_C	Measurement enumeration
DDIF$_SGA_TXT_LEADER_SPACE	Variable
DDIF$_SGA_TXT_LEADER_BULLET	Character string
DDIF$_SGA_TXT_LEADER_ALIGN	Enumeration
DDIF$_SGA_TXT_LEADER_STYLE	Enumeration
DDIF$_SGA_TXT_PAIR_KERNING	Boolean
DDIF$_SGA_LIN_WIDTH_C	Measurement enumeration
DDIF$_SGA_LIN_WIDTH	Variable
DDIF$_SGA_LIN_STYLE	Integer
DDIF$_SGA_LIN_PATTERN_SIZE_C	Measurement enumeration
DDIF$_SGA_LIN_PATTERN_SIZE	Variable
DDIF$_SGA_LIN_MASK_PATTERN	Integer
DDIF$_SGA_LIN_END_START	Enumeration
DDIF$_SGA_LIN_END_FINISH	Enumeration
DDIF$_SGA_LIN_END_SIZE_C	Measurement enumeration
DDIF$_SGA_LIN_END_SIZE	Variable
DDIF$_SGA_LIN_JOIN	Enumeration
DDIF$_SGA_LIN_MITER_LIMIT_N	Integer
DDIF$_SGA_LIN_MITER_LIMIT_D	Integer
DDIF$_SGA_LIN_INTERIOR_PATTERN	Integer
DDIF$_SGA_MKR_STYLE	Enumeration
DDIF$_SGA_MKR_MASK_PATTERN	Integer
DDIF$_SGA_MKR_SIZE_C	Measurement enumeration
DDIF$_SGA_MKR_SIZE	Variable
DDIF$_SGA_GLY_ATTRIBUTES	Handle of DDIF$_GLA aggregate
DDIF$_SGA_IMG_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates
DDIF$_SGA_IMG_PIXEL_PATH	Integer
DDIF$_SGA_IMG_LINE_PROGRESSION	Integer
DDIF$_SGA_IMG_PP_PIXEL_DIST	Integer
DDIF$_SGA_IMG_LP_PIXEL_DIST	Integer
DDIF$_SGA_IMG_BRT_POLARITY	Enumeration
DDIF$_SGA_IMG_GRID_TYPE	Enumeration
DDIF$_SGA_IMG_TIMING_DESC	Binary relative time

(continued on next page)

Table D–23 (Cont.) Segment Attributes Aggregate (DDIF$_SGA)

Item Name	Item Encoding
DDIF$_SGA_IMG_SPECTRAL_MAPPING	Enumeration
DDIF$_SGA_IMG_LOOKUP_TABLES_C	Enumeration
DDIF$_SGA_IMG_LOOKUP_TABLES	Variable
DDIF$_SGA_IMG_COMP_ WAVELENGTH_C	Enumeration
DDIF$_SGA_IMG_COMP_WAVELENGTH	Variable
DDIF$_SGA_IMG_COMP_SPACE_ORG	Enumeration
DDIF$_SGA_IMG_PLANES_PER_PIXEL	Integer
DDIF$_SGA_IMG_PLANE_SIGNIF	Enumeration
DDIF$_SGA_IMG_NUMBER_OF_COMP	Integer
DDIF$_SGA_IMG_BITS_PER_COMP	Array of type integer
DDIF$_SGA_FRM_FLAGS	Longword
DDIF$_SGA_FRM_BOX_LL_X_C	Measurement enumeration
DDIF$_SGA_FRM_BOX_LL_X	Variable
DDIF$_SGA_FRM_BOX_LL_Y_C	Measurement enumeration
DDIF$_SGA_FRM_BOX_LL_Y	Variable
DDIF$_SGA_FRM_BOX_UR_X_C	Measurement enumeration
DDIF$_SGA_FRM_BOX_UR_X	Variable
DDIF$_SGA_FRM_BOX_UR_Y_C	Measurement enumeration
DDIF$_SGA_FRM_BOX_UR_Y	Variable
DDIF$_SGA_FRM_OUTLINE	Sequence of DDIF$_PTH aggregates
DDIF$_SGA_FRM_CLIPPING	Sequence of DDIF$_PTH aggregates
DDIF$_SGA_FRM_POSITION_C	Enumeration
DDIF$_SGA_FRMFXD_POSITION_X_C	Measurement enumeration
DDIF$_SGA_FRMFXD_POSITION_X	Variable
DDIF$_SGA_FRMFXD_POSITION_Y_C	Measurement enumeration
DDIF$_SGA_FRMFXD_POSITION_Y	Variable
DDIF$_SGA_FRMINL_BASE_OFFSET_C	Measurement enumeration
DDIF$_SGA_FRMINL_BASE_OFFSET	Variable
DDIF$_SGA_FRMGLY_VERTICAL	Enumeration
DDIF$_SGA_FRMGLY_HORIZONTAL	Enumeration
DDIF$_SGA_FRMMAR_BASE_OFFSET_C	Measurement enumeration
DDIF$_SGA_FRMMAR_BASE_OFFSET	Variable
DDIF$_SGA_FRMMAR_NEAR_OFFSET_C	Measurement enumeration
DDIF$_SGA_FRMMAR_NEAR_OFFSET	Variable
DDIF$_SGA_FRMMAR_HORIZONTAL	Enumeration

(continued on next page)

Table D–23 (Cont.) Segment Attributes Aggregate (DDIF$_SGA)

Item Name	Item Encoding
DDIF$_SGA_FRM_TRANSFORM	Sequence of DDIF$_TRN aggregates
DDIF$_SGA_ITEM_CHANGE_LIST	Item change list

Table D–24 lists the items in the content definition aggregate and their encodings.

Table D–24 Content Definition Aggregate (DDIF$_CTD)

Item Name	Item Encoding
DDIF$_CTD_LABEL	String
DDIF$_CTD_EXTERNAL_TARGET	String
DDIF$_CTD_EXTERNAL_ERF_INDEX	Integer
DDIF$_CTD_VALUE	Sequence of content
DDIF$_CTD_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates

Table D–25 lists the items in the font definition aggregate and their encodings.

Table D–25 Font Definition Aggregate (DDIF$_FTD)

Item Name	Item Encoding
DDIF$_FTD_NUMBER	Integer
DDIF$_FTD_IDENTIFIER	String
DDIF$_FTD_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates

Table D–26 lists the items in the line style definition aggregate and their encodings.

Table D–26 Line Style Definition Aggregate (DDIF$_LSD)

Item Name	Item Encoding
DDIF$_LSD_NUMBER	Integer
DDIF$_LSD_PATTERN	Array of type integer
DDIF$_LSD_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates

Table D–27 lists the items in the path definition aggregate and their encodings.

Table D–27 Path Definition Aggregate (DDIF$_PHD)

Item Name	Item Encoding
DDIF$_PHD_NUMBER	Integer
DDIF$_PHD_DESCRIPTION	Sequence of DDIF$_PTH aggregates
DDIF$_PHD_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates

Table D–28 lists the items in the pattern definition aggregate and their encodings.

Table D–28 Pattern Definition Aggregate (DDIF$_PTD)

Item Name	Item Encoding
DDIF$_PTD_NUMBER	Integer
DDIF$_PTD_DEFN_C	Enumeration
DDIF$_PTD_SOL_COLOR_C	Enumeration
DDIF$_PTD_SOL_COLOR_R	Single-precision floating-point
DDIF$_PTD_SOL_COLOR_G	Single-precision floating-point
DDIF$_PTD_SOL_COLOR_B	Single-precision floating-point
DDIF$_PTD_PAT_NUMBER	Integer
DDIF$_PTD_PAT_COLORS	Array of type integer
DDIF$_PTD_RAS_PATTERN	Handle of DDIF$_IDU aggregate
DDIF$_PTD_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates

Table D–29 lists the items in the segment binding aggregate and their encodings.

Table D–29 Segment Binding Aggregate (DDIF$_SGB)

Item Name	Item Encoding
DDIF$_SGB_VARIABLE_NAME	String
DDIF$_SGB_VARIABLE_VALUE_C	Enumeration
DDIF$_SGB_CTR_TRIGGER_C	Enumeration
DDIF$_SGB_CTR_TRIGGER	Variable
DDIF$_SGB_CTR_INIT_C	Expression enumeration
DDIF$_SGB_CTR_INIT	Variable
DDIF$_SGB_CTR_STYLE	Sequence of DDIF$_CTS aggregates
DDIF$_SGB_CTR_TYPE	Enumeration
DDIF$_SGB_COM_STRING_EXPR_C	Array of type enumeration

(continued on next page)

Table D–29 (Cont.) Segment Binding Aggregate (DDIF$_SGB)

Item Name	Item Encoding
DDIF$_SGB_COM_STRING_EXPR	Array of type variable
DDIF$_SGB_RCD_LIST	Sequence of DDIF$_RCD aggregates

Table D–30 lists the items in the type definition aggregate and their encodings.

Table D–30 Type Definition Aggregate (DDIF$_TYD)

Item Name	Item Encoding
DDIF$_TYD_LABEL	String
DDIF$_TYD_PARENT	String
DDIF$_TYD_ATTRIBUTES	Handle of DDIF$_SGA aggregate
DDIF$_TYD_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates

Table D–31 lists the items in the counter style aggregate and their encodings.

Table D–31 Counter Style Aggregate (DDIF$_CTS)

Item Name	Item Encoding
DDIF$_CTS_STYLE_C	Enumeration
DDIF$_CTS_STYLE	Variable

Table D–32 lists the items in the occurrence definition aggregate and their encodings.

Table D–32 Occurrence Definition Aggregate (DDIF$_OCC)

Item Name	Item Encoding
DDIF$_OCC_OCCURRENCE_C	Enumeration
DDIF$_OCC_STRUCTURE_ELEMENT_C	Enumeration
DDIF$_OCC_STRUCTURE_ELEMENT	Variable

Table D–33 lists the items in the record definition aggregate and their encodings.

Table D–33 Record Definition Aggregate (DDIF$_RCD)

Item Name	Item Encoding
DDIF$_RCD_TYPE	String

(continued on next page)

Table D–33 (Cont.) Record Definition Aggregate (DDIF$_RCD)

Item Name	Item Encoding
DDIF$_RCD_TAG	String
DDIF$_RCD_CONTENTS	Array of type string

Table D–34 lists the items in the RGB lookup table entry aggregate and their encodings.

Table D–34 RGB Lookup Table Entry Aggregate (DDIF$_RGB)

Item Name	Item Encoding
DDIF$_RGB_LUT_INDEX	Integer
DDIF$_RGB_RED_VALUE	Single-precision floating-point
DDIF$_RGB_GREEN_VALUE	Single-precision floating-point
DDIF$_RGB_BLUE_VALUE	Single-precision floating-point

Table D–35 lists the items in the transformation aggregate and their encodings.

Table D–35 Transformation Aggregate (DDIF$_TRN)

Item Name	Item Encoding
DDIF$_TRN_PARAMETER_C	Enumeration
DDIF$_TRN_PARAMETER	Variable

Table D–36 lists the items in the generic layout 1 aggregate and their encodings.

Table D–36 Generic Layout 1 Aggregate (DDIF$_LG1)

Item Name	Item Encoding
DDIF$_LG1_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates
DDIF$_LG1_PAGE_DESCRIPTIONS	Sequence of DDIF$_PGD aggregates

Table D–37 lists the items in the specific layout 1 aggregate and their encodings.

Table D–37 Specific Layout 1 Aggregate (DDIF$_LS1)

Item Name	Item Encoding
DDIF$_LS1_LAYOUT_C	Array of type enumeration
DDIF$_LS1_LAYOUT	Array of type variable

Table D–38 lists the items in the wrap attributes 1 aggregate and their encodings.

Table D–38 Wrap Attributes 1 Aggregate (DDIF$_LW1)

Item Name	Item Encoding
DDIF$_LW1_WRAP_FORMAT	Enumeration
DDIF$_LW1_QUAD_FORMAT	Enumeration
DDIF$_LW1_HYPHENATION_FLAGS	Longword
DDIF$_LW1_MAXIMUM_HYPH_LINES	Integer
DDIF$_LW1_MAXIMUM_ORPHAN_SIZE	Integer
DDIF$_LW1_MAXIMUM_WIDOW_SIZE	Integer

Table D–39 lists the items in the layout attributes 1 aggregate and their encodings.

Table D–39 Layout Attributes 1 Aggregate (DDIF$_LL1)

Item Name	Item Encoding
DDIF$_LL1_INITIAL_DIRECTIVE	Enumeration
DDIF$_LL1_GALLEY_SELET	String
DDIF$_LL1_BREAK_BEFORE	Enumeration
DDIF$_LL1_BREAK_WITHIN	Enumeration
DDIF$_LL1_BREAK_AFTER	Enumeration
DDIF$_LL1_INITIAL_INDENT_C	Measurement enumeration
DDIF$_LL1_INITIAL_INDENT	Variable
DDIF$_LL1_LEFT_INDENT_C	Measurement enumeration
DDIF$_LL1_LEFT_INDENT	Variable
DDIF$_LL1_RIGHT_INDENT_C	Measurement enumeration
DDIF$_LL1_RIGHT_INDENT	Variable
DDIF$_LL1_SPACE_BEFORE_C	Measurement enumeration
DDIF$_LL1_SPACE_BEFORE	Variable
DDIF$_LL1_SPACE_AFTER_C	Measurement enumeration
DDIF$_LL1_SPACE_AFTER	Variable
DDIF$_LL1_LEADING_RATIO_N	Integer
DDIF$_LL1_LEADING_RATIO_D	Integer
DDIF$_LL1_LEADING_CONSTANT_C	Measurement enumeration
DDIF$_LL1_LEADING_CONSTANT	Variable
DDIF$_LL1_TAB_STOPS	Sequence of DDIF$_TBS aggregates

Table D–40 lists the items in the galley attributes aggregate and their encodings.

Table D–40 Galley Attributes Aggregate (DDIF$_GLA)

Item Name	Item Encoding
DDIF$_GLA_TOP_MARGIN_C	Measurement enumeration
DDIF$_GLA_TOP_MARGIN	Variable
DDIF$_GLA_LEFT_MARGIN_C	Measurement enumeration
DDIF$_GLA_LEFT_MARGIN	Variable
DDIF$_GLA_RIGHT_MARGIN_C	Measurement enumeration
DDIF$_GLA_RIGHT_MARGIN	Variable
DDIF$_GLA_BOTTOM_MARGIN_C	Measurement enumeration
DDIF$_GLA_BOTTOM_MARGIN	Variable

Table D–41 lists the items in the page description aggregate and their encodings.

Table D–41 Page Description Aggregate (DDIF$_PGD)

Item Name	Item Encoding
DDIF$_PGD_LABEL	String
DDIF$_PGD_PRIVATE_DATA	Sequence of DDIF$_PVT aggregates
DDIF$_PGD_DESC_C	Enumeration
DDIF$_PGD_DESC	Variable

Table D–42 lists the items in the page layout aggregate and their encodings.

Table D–42 Page Layout Aggregate (DDIF$_PGL)

Item Name	Item Encoding
DDIF$_PGL_LAYOUT_ID	String
DDIF$_PGL_SIZE_X_NOM_C	Measurement enumeration
DDIF$_PGL_SIZE_X_NOM	Variable
DDIF$_PGL_SIZE_X_STR_C	Measurement enumeration
DDIF$_PGL_SIZE_X_STR	Variable
DDIF$_PGL_SIZE_X_SHR_C	Measurement enumeration
DDIF$_PGL_SIZE_X_SHR	Variable
DDIF$_PGL_SIZE_Y_NOM_C	Measurement enumeration
DDIF$_PGL_SIZE_Y_NOM	Variable
DDIF$_PGL_SIZE_Y_STR_C	Measurement enumeration
DDIF$_PGL_SIZE_Y_STR	Variable
DDIF$_PGL_SIZE_Y_SHR_C	Measurement enumeration

(continued on next page)

Table D–42 (Cont.) Page Layout Aggregate (DDIF$_PGL)

Item Name	Item Encoding
DDIF$_PGL_SIZE_Y_SHR	Variable
DDIF$_PGL_ORIENTATION	Enumeration
DDIF$_PGL_PROTOTYPE	String
DDIF$_PGL_CONTENT	Sequence of content

Table D–43 lists the items in the page select aggregate and their encodings.

Table D–43 Page Select Aggregate (DDIF$_PGS)

Item Name	Item Encoding
DDIF$_PGS_PAGE_SIDE_CRITERIA	Enumeration
DDIF$_PGS_SELECT_PAGE_LAYOUT_C	Enumeration
DDIF$_PGS_SELECT_PAGE_LAYOUT	Variable

Table D–44 lists the items in the tab stop aggregate and their encodings.

Table D–44 Tab Stop Aggregate (DDIF$_TBS)

Item Name	Item Encoding
DDIF$_TBS_HORIZONTAL_POSITION_C	Measurement enumeration
DDIF$_TBS_HORIZONTAL_POSITION	Variable
DDIF$_TBS_TYPE	Enumeration
DDIF$_TBS_LEADER	Character string

E DDIF Syntax Diagrams

This appendix lists the syntax diagrams for each construct defined by the DIGITAL Document Interchange Format. The abstract syntax notation used to define these constructs at the lowest level is the DIGITAL Data Interchange Syntax (DDIS). The elements of the DDIS abstract syntax notation that are used in this appendix are summarized in the following sections.

E.1 DDIS Built-In Data Types

Table E–1 lists the built-in types that are primitive data types:

Table E–1 DDIS Built-In Primitives

Type	Definition
NULL	A data element with no value
INTEGER	A signed, two's complement binary number
BOOLEAN	A Boolean value, constrained to be true or false
BIT STRING	A string of bits
OCTET STRING	A character string or other data type that logically consists of a series of "octet" (8-bit quantity) values
FLOATING-POINT	An element that consists of a sign magnitude, with bit 7 of the second octet representing the sign bit. Bits 6 through 0 of the second octet and bits 7 through 0 of the first octet collectively encode an excess-16384 binary exponent. The bits of the exponent decrease in significance from bit 6 to bit 0 of the second octet, and then from bit 7 to bit 0 of the first octet. The remaining (zero or more) octets of the value encode a normalized fraction with the redundant most significant bit not represented. The fraction is encoded such that bits increase in significance from bit 0 through bit 15 of each octet pair, and successive pairs of octets become less significant.

(continued on next page)

Table E–1 (Cont.) DDIS Built-In Primitives

Type	Definition
OBJECT IDENTIFIER	A list of object identifier components, which are integer values that identify branches in a tree of object identifiers. The value field of an element of type OBJECT IDENTIFIER consists of an ordered list of subidentifiers, where each subidentifier is an unsigned integer value. Each subidentifier is represented as one or more octets. If bit 7 of a given octet is set, the subidentifier is continued in the next octet. Bits 6 through 0 of the octets in the subidentifier collectively encode an integer that represents a branch in the registration tree. These bits are concatenated to form an unsigned integer whose most significant bit is bit 6 of the first octet and whose least significant bit is bit 0 of the last octet.
EXTERNAL	A data value whose basic encoding may or may not conform to the DIGITAL Data Interchange Syntax. The direct-reference element in the EXTERNAL data type indicates the data type (syntax and semantics) of the external element. The data-value descriptor element is a text string that describes the data value in a human-readable form. The encoding field contains the data value itself.

The DDIF syntax diagrams also refer to a Generalized Time universal defined type. This type represents a calendar date and time of day to various precisions. The time of day can be specified as local time only, as Coordinated Universal Time (UTC) only, or as both local and UTC.

The Generalized Time type represents time by a string of characters consisting of:

- A calendar date
- A time of day
- The local Time Differential Factor (TDF)

In addition to these primitive data types, DDIS also provides built-in constructors (records and arrays). Table E–2 shows the DDIS constructors used in the DDIF syntax diagrams.

Table E–2 DDIS Built-In Constructors

Constructor	Definition
SEQUENCE	A list of elements that can be primitive or themselves constructed, which must occur in the order in which the elements are specified. A SEQUENCE can be viewed as a record in which each field has a type identifier in the data stream. All elements of a SEQUENCE are enclosed within braces.
SEQUENCE OF	A list of elements that can be primitive or themselves constructed, which are all of a specified type. For example, a "SEQUENCE OF INTEGER" models a list of integers.

DDIS also provides **tagged types**. Elements in the syntax are often assigned tags for the purpose of making them unique within their context. These tags, shown in the syntax as a number between square brackets, serve to identify the element. Note that they are not counters; while they are conventionally assigned in ascending order to elements of a constructor type, they are not constrained to do so. Elements of a SEQUENCE occur in the order in which they are listed.

Tagged types can use the IMPLICIT keyword to specify that the tagged type assumes the encoding of the referenced type, rather than forming a constructor containing a built-in element. Use of the IMPLICIT keyword reduces the number of bytes required to represent the encoded data, but requires that decoding software have knowledge of the type.

E.2 Built-In Operators

Table E–3 describes the DDIS built-in operators. They are best described as operators because they affect the way the built-in types are encoded. The keywords for built-in operators are expressed in uppercase letters.

Table E–3 DDIS Built-In Operators

Operator	Effects
CHOICE	Only one of the list of alternative types can be chosen. Note that CHOICE is not a type that has a tag. It therefore cannot be preceded by the IMPLICIT operator. CHOICE can force a tagged type to become a constructor that then contains the chosen alternative.
OPTIONAL	The designated element can be omitted at the option of the sending application.
DEFAULT	The designated element has a default value. Elements with default values are also optional and can be omitted at the option of the sending application. The receiving application uses the specified default value when the element is missing from the encoding.

(continued on next page)

Table E–3 (Cont.) DDIS Built-In Operators

Operator	Effects
ANY	Any tagged element can be inserted in the encoding, at the option of the sending application.
Assignment	The assignment operator, represented by two colons and an equal sign (::=), assigns a name to a syntax definition by which it can be referenced in other definitions. Elements of a syntax can therefore share a definition.
Named number	The assignment of an identifier to a specific value. Named numbers are often used for clarity in referring to values with specific meaning, and to provide for automatic generation of symbolic values for use in software development. (By convention, named integer values in DDIF start from 1 and named bits start from bit 0.)
Comments	The comment delimiter, represented by two consecutive hyphen characters (- -), causes the text following this delimiter to be treated as a comment.

E.3 DDIS Defined Types

Table E–4 shows the types defined by DDIS:

Table E–4 DDIS Defined Types

Defined Type	Encoding
Latin1-String	An element encoded as an OCTET STRING in which all octet values represent characters from the Latin1 character set. Characters 32 through 126 of this character set are the same as the 7-bit ASCII code.
Character-String	An element in which the first octet or octets identify the character set, and the remaining octets constitute the codes of characters selected from that character set. The characters in a Character-String type can be chosen from 8-bit, 16-bit, and 32-bit character sets.
Text-String	An element that consists of a sequence of Character-String elements, and can thus represent a text string in which characters are selected from more than one character set.

E.4 DDIF Syntax Diagrams

This section lists all of the syntax diagrams that are used to describe the DIGITAL Document Interchange Format constructs. Figure E–1 illustrates the syntax used to create a DDIF document construct.

Figure E–1 DDIF Document Syntax Diagram

```
DDIFDocument                ::= [PRIVATE 16383] IMPLICIT SEQUENCE {
     document-descriptor          [0] IMPLICIT DocumentDescriptor,
     document-header              [1] IMPLICIT DocumentHeader,
     document-content             [2] IMPLICIT Content
                                  }
```

Figure E–2 illustrates the syntax used to create a document descriptor construct.

Figure E–2 Document Descriptor Syntax Diagram

```
DocumentDescriptor          ::= SEQUENCE {
    major-version                [0] IMPLICIT INTEGER,
    minor-version                [1] IMPLICIT INTEGER,
    product-identifier           [2] IMPLICIT ASCIIString,
    product-name                 [3] IMPLICIT Text-String
                                 }
```

Figure E–3 illustrates the syntax used to create a document header construct.

Figure E–3 Document Header Syntax Diagram

```
DocumentHeader              ::=  SEQUENCE {
     private-header-data          [0] IMPLICIT NamedValueList  OPTIONAL,
     title                        [1] IMPLICIT Text-String     OPTIONAL,
     author                       [2] IMPLICIT Text-String     OPTIONAL,
     version                      [3] IMPLICIT Text-String     OPTIONAL,
     date                         [4] IMPLICIT GeneralizedTime OPTIONAL,
     conformance-tags             [5] IMPLICIT SEQUENCE OF ConformanceTag OPTIONAL,
     external-references          [6] IMPLICIT SEQUENCE OF ExternalReference
                                      OPTIONAL,
     languages                    [7] IMPLICIT SEQUENCE OF CHOICE {
          iso-639-language                [0] IMPLICIT ASCIIString,
          other-language                  [1] IMPLICIT Character-String
                                          }   OPTIONAL,
     style-guide                  [8] IMPLICIT ExternalRefIndex OPTIONAL
                                  }
```

Figure E–4 illustrates the syntax used to create a document root segment construct.

Figure E–4 Document Root Segment

```
Content                    ::= SEQUENCE OF ContentPrimitive

ContentPrimitive           ::= CHOICE {
     segment-primitive           SegmentPrimitive,
     text-primitive              TextPrimitive,
     formatting-primitive        FormattingPrimitive,
     graphics-primitive          GraphicsPrimitive,
     image-primitive             ImagePrimitive,
     content-ref-primitive       ContentReferencePrimitive,
     restricted-content          RestrictedContent,
     layout-primitive            LayoutPrimitive
     }
```

Figure E–5 illustrates the syntax used to create a segment primitive construct.

Figure E–5 Segment Primitive Syntax Diagram

```
SegmentPrimitive                ::= CHOICE {
     end-segment                        [APPLICATION 1] IMPLICIT NULL,
     begin-segment                      [APPLICATION 2] IMPLICIT BeginSegment
                                        }
```

Figure E–6 illustrates the syntax used to create a construct.

Figure E–6 Begin-Segment Syntax Diagram

```
BeginSegment               ::= SEQUENCE {
     segment-id                  [0] IMPLICIT SegmentLabel      OPTIONAL,
     user-label                  [1] IMPLICIT Text-String       OPTIONAL,
     segment-type                [2] IMPLICIT TypeDefnLabel     OPTIONAL,
     specific-attributes         [3] IMPLICIT SegmentAttributes OPTIONAL,
     generic-layout              [4] GenericLayout              OPTIONAL, --  ANY
     specific-layout             [5] SpecificLayout             OPTIONAL  --  ANY
                                 }
```

Figure E–7 illustrates the syntax used to create a text primitive construct.

Figure E–7 Text Primitive Syntax Diagram

```
TextPrimitive              ::= CHOICE {
     latin1-content                [APPLICATION 3] IMPLICIT Latin1-String,
     general-text-content          [APPLICATION 4] IMPLICIT Character-String
                                   }
```

Figure E–8 illustrates the syntax used to create a text attributes construct.

Figure E–8 Text Attributes Syntax Diagram

```
TextAttributes             ::= SEQUENCE {
     text-mask-pattern             [0] IMPLICIT PatternNumber  OPTIONAL,
     text-font                     [1] IMPLICIT FontNumber     OPTIONAL,
     text-rendition                [2] IMPLICIT SEQUENCE OF
                                       RenditionCode           OPTIONAL,
     text-height                   [3] Size                    OPTIONAL,
     text-set-size                 [4] IMPLICIT Ratio          OPTIONAL,
     text-direction                [5] IMPLICIT INTEGER {
          text-dir-forward(1),
          text-dir-backward(2)           }                     OPTIONAL,
     decimal-align-chars           [6] IMPLICIT SEQUENCE OF
                                       Character-String        OPTIONAL,
     leader-attributes             [7] IMPLICIT LeaderStyle    OPTIONAL,
     pair-kerning                  [8] IMPLICIT BOOLEAN        OPTIONAL
                                   }
```

Figure E–9 illustrates the syntax used to create a rendition code construct.

Figure E–9 Rendition Code Syntax Diagram

```
RenditionCode              ::= INTEGER {
     default(0),
     highlighted(1),
     faint(2),
     italic(3),
     underlined(4),
     slow-blink(5),
     rapid-blink(6),
     negative-image(7),
     concealed-chars(8),
     crossed-out(9),
     double-underlined(21),
     normal-intensity(22),
     not-underlined(24),
     steady(25),
     positive(27),
     revealed-chars(28),
     boxed(51),
     encircled(52),
     overlined(53),
     ideogram-underlined(60),
     ideogram-db-underlined(61),
```

(continued on next page)

Figure E–9 (Cont.) Rendition Code Syntax Diagram

```
     ideogram-overlined(62),
     ideogram-db-overlined(63),
     ideogram-stress-mark(64)            }
```

Figure E–10 illustrates the syntax used to create a leader style construct.

Figure E–10 Leader Style Syntax Diagram

```
LeaderStyle               ::= SEQUENCE {
     leader-space              [0] Size                       OPTIONAL,
     leader-bullet             [1] IMPLICIT Character-String OPTIONAL,
     leader-align              [2] IMPLICIT INTEGER {
          aligned-leader(1),
          staggered-leader(2),
          non-aligned-leader(3) }                             OPTIONAL,
     leader-style              [3] IMPLICIT INTEGER {
          ls-x-rule(1),
          ls-bullet(2)                              }        OPTIONAL
                                }
```

Figure E–11 illustrates the syntax used to create a text layout construct.

Figure E–11 Text Layout Syntax Diagram

```
TextLayout             ::= CHOICE {
     galley-based-layout       [0] IMPLICIT SEQUENCE {
          wrap-attributes            [0] WrapAttributes OPTIONAL, -- Defined as ANY
          galley-layout              [1] GalleyLayout   OPTIONAL  -- Defined as ANY
                                     },
     path-based-layout         [1] IMPLICIT StringLayout,
     position-relative         [2] IMPLICIT SEQUENCE {
         vertical-offset                [0] IMPLICIT Escapement OPTIONAL,
         horizontal-offset              [1] IMPLICIT Escapement OPTIONAL
                                        },
     text-position             [3] IMPLICIT INTEGER {
          tp-base(1),
          tp-left-subscript(2),
          tp-left-superscript(3),
          tp-right-subscript(4),
          tp-right-superscript(5),
          tp-top-center(6),
          tp-bottom-center(7),
          tp-rubi(8)                                 }
                                }
```

Figure E–12 illustrates the syntax used to create a text string layout construct.

Figure E–12 Text String Layout Syntax Diagram

```
StringLayout              ::= SEQUENCE {
     string-layout-path         [0] IMPLICIT CompositePath,
     string-layout-format       [1] IMPLICIT Format DEFAULT flush-path-begin,
     character-orientation      CHOICE {
          char-angle-fixed           [2] IMPLICIT Angle,
          char-angle-path            [3] IMPLICIT RightAngle
                                     }   DEFAULT { char-angle-path up },
     char-horizontal-align      [4] IMPLICIT INTEGER {
          normal-horizontal(1),
          leftline(2),
          centerline(3),
          rightline(4)   }          DEFAULT normal-horizontal,
     char-vertical-align        [5] IMPLICIT INTEGER {
          normal-vertical(1),
          baseline(2),
          capline(3),
          bottomline(4),
          halfline(5),
          topline(6)  }             DEFAULT normal-vertical
                                  }
```

Figure E–13 illustrates the syntax used to create a formatting primitive construct.

Figure E–13 Formatting Primitive Syntax Diagram

```
FormattingPrimitive       ::= CHOICE {
     soft-value-directive       [APPLICATION 7] ValueDirective,
     hard-value-directive       [APPLICATION 8] ValueDirective,
     hard-directive             [APPLICATION 9] IMPLICIT Directive,
     soft-directive            [APPLICATION 10] IMPLICIT Directive
                                }
```

Figure E–14 illustrates the syntax used to create a value directive construct.

Figure E–14 Value Directive Syntax Diagram

```
ValueDirective              ::= CHOICE {
     escapement-directive         [0] IMPLICIT EscapementDirective,
     variable-reset               [1] IMPLICIT VariableReset
                                  }
```

Figure E–15 illustrates the syntax used to create a directive construct.

Figure E–15 Directive Syntax Diagram

```
Directive                   ::= INTEGER {
     new-page(1),
     new-line(2),
     new-galley(3),
     tab(4),
     space(5),
     hyphen-new-line(6),
     word-break-point(7),
     leaders(8),
     backspace(9),
     null-directive(10),
     no-hyphen-word(11)     }
```

Figure E–16 illustrates the syntax used to create an escapement directive construct.

Figure E–16 Escapement Directive Syntax Diagram

```
EscapementDirective          ::= Escapement
```

Figure E–17 illustrates the syntax used to create a variable reset construct.

Figure E–17 Variable Reset Syntax Diagram

```
VariableReset ::= SEQUENCE {
     reset-variable      [0] IMPLICIT VariableLabel,
     reset-value         [1] Expression
                         }
```

Figure E–18 illustrates the syntax used to create a graphics primitive construct.

Figure E–18 Graphics Primitive Syntax Diagram

```
GraphicsPrimitive          ::= CHOICE {
     cubic-curve-object          [APPLICATION 11] IMPLICIT CubicBezier,
     polyline-object             [APPLICATION 12] IMPLICIT Polyline,
     arc-object                  [APPLICATION 13] IMPLICIT Arc,
     fill-area-set               [APPLICATION 14] IMPLICIT FillAreaSet
                                 }
```

Figure E–19 illustrates the syntax used to create a polyline construct.

Figure E–19 Polyline Syntax Diagram

```
Polyline                   ::= SEQUENCE {
     polyline-flags                [0] IMPLICIT BIT STRING {
          draw-polyline(0),
          fill-polyline(1),
          draw-markers(2),
          regular-polygon(3),
          close-polyline(4),
          rounded-polyline(5),
          rectangular-polygon(6) } DEFAULT { draw-polyline },
     polyline-draw-pattern         [1] IMPLICIT BIT STRING DEFAULT '1'B,
     polyline-path                 [2] IMPLICIT PolyLinePath
                                   }
```

E.4 DDIF Syntax Diagrams

Figure E–20 illustrates the syntax used to create a cubic Bézier construct.

Figure E–20 Cubic Bézier Syntax Diagram

```
CubicBezier                ::= SEQUENCE {
     cubic-Bezier-flags          [0] IMPLICIT BIT STRING {
          draw-cb(0),
          fill-cb(1),
          close-cb(2)   }             DEFAULT { draw-cb },
     cubic-Bezier-path           [1] IMPLICIT CubicBezierPath
                                 }
```

Figure E–21 illustrates the syntax used to create an arc construct.

Figure E–21 Arc Syntax Diagram

```
Arc                        ::=  SEQUENCE {
     arc-flags                    [0] IMPLICIT BIT STRING {
          draw-arc(0),
          fill-arc(1),
          pie-arc(2),
          close-arc(3)   }            DEFAULT { draw-arc },
     arc-path                     [1] IMPLICIT ArcPath
                                  }
```

Figure E–22 illustrates the syntax used to create a fill area set construct.

Figure E–22 Fill Area Set Syntax Diagram

```
FillAreaSet                ::= SEQUENCE {
     fas-flags                    [0] IMPLICIT BIT STRING {
          co-draw-border(0),
          co-fill-area(1) }           DEFAULT { co-draw-border },
     fas-path                     [1] IMPLICIT CompositePath
                                  }
```

Figure E–23 illustrates the syntax used to create a line attributes construct.

Figure E–23 Line Attributes Syntax Diagram

```
LineAttributes              ::= SEQUENCE {
     line-width                   [0] Size                        OPTIONAL,
     line-style                   [1] IMPLICIT LineStyleNumber    OPTIONAL,
     line-pattern-size            [2] Size                        OPTIONAL,
     line-mask-pattern            [3] IMPLICIT PatternNumber      OPTIONAL,
     line-end-start               [4] IMPLICIT LineEndNumber      OPTIONAL,
     line-end-finish              [5] IMPLICIT LineEndNumber      OPTIONAL,
     line-end-size                [6] Size                        OPTIONAL,
     line-join                    [7] IMPLICIT LineJoin           OPTIONAL,
     line-miter-limit             [8] IMPLICIT Ratio              OPTIONAL,
     line-interior-pattern        [9] IMPLICIT PatternNumber      OPTIONAL
                                  }
```

Figure E–24 illustrates the syntax used to create a line style number construct.

Figure E–24 Line Style Number Syntax Diagram

```
LineStyleNumber             ::= INTEGER {
     solid(1),
     dash(2),
     dot(3),
     dash-dot(4)
                                          }
```

Figure E–25 illustrates the syntax used to create a line end number construct.

Figure E–25 Line End Number Syntax Diagram

```
LineEndNumber               ::= INTEGER {
     butt-line-end(1),
     round-line-end(2),
     square-line-end(3),
     arrow(4)                             }
```

Figure E–26 illustrates the syntax used to create a line join construct.

Figure E–26 Line Join Syntax Diagram

```
LineJoin                    ::= INTEGER {
     mitered-line-join(1),
     rounded-line-join(2),
     beveled-line-join(3)
                                    }
```

Figure E–27 illustrates the syntax used to create a marker attributes construct.

Figure E–27 Marker Attributes Syntax Diagram

```
MarkerAttributes                ::= SEQUENCE {
     marker-style                     [0] IMPLICIT MarkerNumber  OPTIONAL,
     marker-mask-pattern              [1] IMPLICIT PatternNumber OPTIONAL,
     marker-size                      [2] Size                   OPTIONAL
                                      }
```

Figure E–28 illustrates the syntax used to create a marker number construct.

Figure E–28 Marker Number Syntax Diagram

```
MarkerNumber                ::= INTEGER {
     marker-dot(1),
     marker-plus-sign(2),
     marker-asterisk(3),
     marker-circle(4),
     marker-diagonal-cross(5)
     }
```

Figure E–29 illustrates the syntax used to create an image primitive construct.

Figure E–29 Image Primitive Syntax Diagram

```
ImagePrimitive              ::= CHOICE {
       image-content                [APPLICATION 17] IMPLICIT ImageDataDescriptor
                                    }

ImageDataDescriptor         ::= SEQUENCE OF ImageDataUnit

ImageDataUnit               ::= SEQUENCE {
        image-coding-attrs           [0] IMPLICIT ImageCodingAttrs,
        image-comp-plane-data        [1] IMPLICIT OCTET STRING
                                     }
```

Figure E–30 illustrates the syntax used to create an image coding attributes construct.

Figure E–30 Image Coding Attributes Syntax Diagram

```
ImageCodingAttrs            ::= SEQUENCE {
     pvt-img-coding-attrs         [0] IMPLICIT NamedValueList OPTIONAL,
     pixels-per-line              [1] IMPLICIT INTEGER,
     number-of-lines              [2] IMPLICIT INTEGER,
     compression-type             [3] IMPLICIT INTEGER {
          private-compression      (1),
          pcm-compression          (2),       -- (raw bitmap)
          g31d-compression         (3),       -- CCITT Group 3 1 dimensional
          g32d-compression         (4),       -- CCITT Group 3 2 dimensional
          g42d-compression         (5)        -- CCITT Group 4 2 dimensional
                                   } DEFAULT pcm-compression,
     compression-parameters       [4] IMPLICIT NamedValueList OPTIONAL,
     data-offset                  [5] IMPLICIT INTEGER        DEFAULT 0,
     pixel-stride                 [6] IMPLICIT INTEGER        OPTIONAL,
     scanline-stride              [7] IMPLICIT INTEGER        OPTIONAL,
     pixel-order                  [8] IMPLICIT INTEGER {
          standard-pixel-order (1),
          reverse-pixel-order (2) } DEFAULT standard-pixel-order,
     planebits-per-pixel          [9] IMPLICIT INTEGER OPTIONAL
                                  }
```

Figure E–31 illustrates the syntax used to create an image attributes construct.

Figure E–31 Image Attributes Syntax Diagram

```
ImageAttributes           ::= SEQUENCE {
     img-present-attrs         [0] IMPLICIT ImgPresentAttrs OPTIONAL,
     img-comp-space-attrs      [1] IMPLICIT ImgCmptSpcAttrs OPTIONAL
                               }

ImgPresentAttrs           ::= SEQUENCE {
     prvt-img-present-attrs    [0] IMPLICIT NamedValueList OPTIONAL,
     pixel-path                [1] IMPLICIT INTEGER        OPTIONAL,
     line-progression          [2] IMPLICIT INTEGER        OPTIONAL,
     pixel-aspect-ratio        [3] IMPLICIT SEQUENCE {
          pxl-path-pxl-distance     [0] IMPLICIT INTEGER   DEFAULT 1,
          line-prog-pxl-distance    [1] IMPLICIT INTEGER   DEFAULT 1
                                    }                      OPTIONAL,
     brightness-polarity       [4] IMPLICIT INTEGER {
          zero-maximum-intensity(1),
          zero-minimum-intensity(2)  }                     OPTIONAL,

     grid-type                 [5] IMPLICIT INTEGER {
          rectangular-grid(1),
          hex-even-indent(2),
          hex-odd-indent(3)  }     OPTIONAL,

     timing-descriptor         [6] IMPLICIT Binary-Relative-Time OPTIONAL,
     spectral-comp-mapping     [7] IMPLICIT INTEGER {
          privately-mapped (1),
          monochrome-mapped (2),
          general-multispectral (3),
          lut-mapped (4),
          rgb-mapped (5),
          cmy-mapped (6),
          yuv-mapped (7),
          hsv-mapped (8),
          hls-mapped (9),
          yiq-mapped (10) }        OPTIONAL,

     lookup-tables             [8] ImgLutData OPTIONAL,
     component-wlength-info    [9] CHOICE {
          application-wlen-info     [0] IMPLICIT SEQUENCE OF OCTET STRING,
          wavelength-measure        [1] IMPLICIT SEQUENCE OF INTEGER,
          wavelength-band-id        [2] IMPLICIT SEQUENCE OF Latin1-String
                                    }   OPTIONAL
                               }
```

Figure E–32 illustrates the syntax used to create an image lookup table data construct.

Figure E–32 Image Lookup Table Data Syntax Diagram

```
ImgLutData                  ::= CHOICE {
    application-pvt-luts            [0] IMPLICIT NamedValueList,
    rgb-lut-entries                 [1] IMPLICIT SEQUENCE OF RgbLutEntry
                                    }

RgbLutEntry                 ::= SEQUENCE {
    lut-index                       [0] IMPLICIT INTEGER,
    red-value                       [1] IMPLICIT ColorIntensity,
    green-value                     [2] IMPLICIT ColorIntensity,
    blue-value                      [3] IMPLICIT ColorIntensity
                                    }
```

Figure E–33 illustrates the syntax used to create an image component space attributes construct.

Figure E–33 Image Component Space Attributes Syntax Diagram

```
ImgCmptSpcAttrs           ::= SEQUENCE {
     comp-space-org               [0] IMPLICIT INTEGER {
          full-compaction(1),
          partial-expansion(2),
          full-expansion(3) }                                    OPTIONAL,

     data-planes-per-pixel        [1] IMPLICIT INTEGER           OPTIONAL,
     data-plane-signif            [2] IMPLICIT INTEGER {
           lsb-msb (1),
           msb-lsb (2)  }                                        OPTIONAL,

     number-of-components         [3] IMPLICIT INTEGER,
     bits-per-component-lst       [4] IMPLICIT SEQUENCE OF INTEGER
                                  }
```

Figure E–34 illustrates the syntax used to create a restricted content construct.

Figure E–34 Restricted Content Syntax Diagram

```
  RestrictedContent ::= CHOICE {
      pdl-content       [APPLICATION 18] IMPLICIT EXTERNAL,
      private-content   [APPLICATION 30] IMPLICIT NamedValue
                        }
```

Figure E–35 illustrates the syntax used to create a content reference primitive construct.

Figure E–35 Content Reference Primitive Syntax Diagram

```
ContentReferencePrimitive ::= CHOICE {
     content-ref                [APPLICATION 15] IMPLICIT ContentReference
                                }
```

Figure E–36 illustrates the syntax used to create a content reference construct.

Figure E–36 Content Reference Syntax Diagram

```
ContentReference          ::= SEQUENCE {
     content-transform          [0] IMPLICIT Transformation OPTIONAL,
     content-reference          [1] IMPLICIT ContentDefnLabel
                                }
```

Figure E–37 illustrates the syntax used to create a bounding box construct.

Figure E–37 Bounding Box Syntax Diagram

```
BoundingBox               ::= SEQUENCE {
     lower-left                 [0] IMPLICIT Position,
     upper-right                [1] IMPLICIT Position
                                }
```

Figure E–38 illustrates the syntax used to create a color construct.

Figure E–38 Color Syntax Diagram

```
Color                     ::= CHOICE {
    rgb-color                   [0] IMPLICIT RGB,
    transparency                [1] IMPLICIT NULL
                                }
```

Figure E–39 illustrates the syntax used to create a red/green/blue construct.

Figure E–39 Red/Green/Blue Syntax Diagram

```
RGB                     ::= SEQUENCE {
    red-intensity            [0] IMPLICIT ColorIntensity DEFAULT 0.0,
    green-intensity          [1] IMPLICIT ColorIntensity DEFAULT 0.0,
    blue-intensity           [2] IMPLICIT ColorIntensity DEFAULT 0.0
                             }

ColorIntensity          ::= FLOATING-POINT
```

Figure E–40 illustrates the syntax used to create a compute definition construct.

Figure E–40 Compute Definition Syntax Diagram

```
ComputeDefn                 ::= CHOICE {
     copy-content                [0] IMPLICIT Reference,
     remote-content              [1] IMPLICIT Reference,
     variable-reference          [2] IMPLICIT VariableLabel,
     cross-reference             [3] IMPLICIT CrossReference,
     function-link               [4] IMPLICIT FunctionLink
                                 }
```

Figure E–41 illustrates the syntax used to create a cross-reference construct.

Figure E–41 Cross Reference Syntax Diagram

```
CrossReference              ::= SEQUENCE {
     xref-seg-label              [0] IMPLICIT Reference,
     xref-var-label              [1] IMPLICIT VariableLabel
                                 }
```

Figure E–42 illustrates the syntax used to create an escapement construct.

Figure E–42 Escapement Syntax Diagram

```
Escapement                ::= SEQUENCE {
     escapement-ratio           [0] IMPLICIT Ratio OPTIONAL,
     escapement-constant        [1] Measurement    OPTIONAL
                                }
```

Figure E–43 illustrates the syntax used to create an external reference construct.

Figure E–43 External Reference Syntax Diagram

```
ExternalReference         ::= SEQUENCE {
     reference-data-type        [0] IMPLICIT OBJECT IDENTIFIER,
     reference-descriptor       [1] IMPLICIT Text-String,
     reference-label            [2] IMPLICIT Character-String,
     reference-label-type       [3] IMPLICIT StorageSystemTag,
     reference-control          [4] IMPLICIT INTEGER {
          copy-reference(1),
          no-copy-reference(2)
          }                         DEFAULT copy-reference
                                }
```

Figure E–44 illustrates the syntax used to create a font definition construct.

Figure E–44 Font Definition Syntax Diagram

```
FontDefn                  ::= SEQUENCE {
     font-number                [0] IMPLICIT FontNumber,
     font-identifier            [1] IMPLICIT Latin1-String,
     font-private               [2] IMPLICIT NamedValueList OPTIONAL
                                }
```

Figure E–45 illustrates the syntax used to create a format construct.

Figure E–45 Format Syntax Diagram

```
Format        ::= INTEGER { flush-path-begin(1), center-of-path(2),
                            flush-path-end(3), flush-path-both(4)
                            }
```

Figure E–46 illustrates the syntax used to create a frame parameters construct.

Figure E–46 Frame Parameters Syntax Diagram

```
FrameParameters             ::= SEQUENCE {
     frame-flags                    [0] IMPLICIT BIT STRING {
          flow-around (0),
          frame-border(1),
          frame-background-fill(2) }                        OPTIONAL,
     frame-bounding-box             [1] IMPLICIT BoundingBox,
     frame-outline                  [2] IMPLICIT CompositePath  OPTIONAL,
     frame-clipping                 [3] IMPLICIT CompositePath  OPTIONAL,
     frame-position                 CHOICE {
          fp-fixed                       [4] IMPLICIT Position,
          fp-inline                      [5] IMPLICIT InlineFrameParams,
          fp-galley                      [6] IMPLICIT GalleyFrameParams,
          fp-margin                      [7] IMPLICIT MarginFrameParams
                                         },
     frame-content-trans            [8] IMPLICIT Transformation OPTIONAL
                                    }
```

Figure E–47 illustrates the syntax used to create an inline frame parameters construct.

Figure E–47 Inline Frame Parameters Syntax Diagram

```
InlineFrameParams           ::= SEQUENCE {
     ifp-base-offset                [0] Size DEFAULT { integer-constant 0 }
                                    }
```

Figure E–48 illustrates the syntax used to create a galley frame parameters construct.

Figure E–48 Galley Frame Parameters Syntax Diagram

```
GalleyFrameParams          ::= SEQUENCE {
     gfp-vertical                [0] IMPLICIT GalleyVerticalPosition
                                     DEFAULT below-current-line,
     gfp-horizontal              [1] IMPLICIT Format DEFAULT center-of-path
                                 }
```

Figure E–49 illustrates the syntax used to create a galley vertical position construct.

Figure E–49 Galley Vertical Position Syntax Diagram

```
GalleyVerticalPosition ::= INTEGER {
     below-current-line(1),
     bottom-of-galley(2),
     top-of-galley(3)
     }
```

Figure E–50 illustrates the syntax used to create a margin frame parameters construct.

Figure E–50 Margin Frame Parameters Syntax Diagram

```
MarginFrameParams          ::= SEQUENCE {
     mfp-base-offset             [0] Size DEFAULT { integer-constant 0 },
     mfp-near-offset             [1] Size DEFAULT { integer-constant 0 },
     mfp-horizontal              [2] IMPLICIT MarginHorizontalPosition
                                     DEFAULT side-closest-edge
                                 }
```

Figure E–51 illustrates the syntax used to create a margin horizontal position construct.

Figure E–51 Margin Horizontal Position Syntax Diagram

```
MarginHorizontalPosition ::= INTEGER {
     side-closest-edge(1),
     side-furthest-edge(2),
     left-of-galleys(3),
     right-of-galleys(4)
     }
```

Figure E–52 illustrates the syntax used to create a function link construct.

Figure E–52 Function Link Syntax Diagram

```
FunctionLink                ::= SEQUENCE {
     function-name                   [0] IMPLICIT ASCIIString,
     function-parameters             [1] IMPLICIT NamedValueList
                                     }
```

Figure E–53 illustrates the syntax used to create an external reference index construct.

Figure E–53 External Reference Index Syntax Diagram

```
ExternalRefIndex ::= INTEGER
```

Figure E–54 illustrates the syntax used to create a language index construct.

Figure E–54 Language Index Syntax Diagram

```
LanguageIndex ::= INTEGER
```

Figure E–55 illustrates the syntax used to create a content definition construct.

Figure E–55 Content Definition Syntax Diagram

```
ContentDefn                ::= SEQUENCE {
     content-label                 [0] IMPLICIT ContentDefnLabel,
     content-external              [1] IMPLICIT Reference      OPTIONAL,
     content-value                 [2] IMPLICIT Content        OPTIONAL,
     content-private               [3] IMPLICIT NamedValueList OPTIONAL
                                   }
```

Figure E–56 illustrates the syntax used to create a label construct.

Figure E–56 Label Syntax Diagram

```
Label                   ::= ASCIIString
```

Figure E–57 illustrates the syntax used to create a label types construct.

Figure E–57 Label Types Syntax Diagram

```
VariableLabel           ::= Label
SegmentLabel            ::= Label
TypeDefnLabel           ::= Label
ContentDefnLabel        ::= Label
GalleyLabel             ::= Label
PageDescLabel           ::= Label
PageLayoutLabel         ::= Label
```

Figure E–58 illustrates the syntax used to create an ASCII string construct.

Figure E–58 ASCII String Syntax Diagram

```
ASCIIString ::= Latin1-String
```

Figure E–59 illustrates the syntax used to create a variable label construct.

Figure E–59 Variable Label Syntax Diagram

```
VariableLabel          ::= Label       -- used to refer to variable by name
```

Figure E–60 illustrates the syntax used to create a legend units construct.

Figure E–60 Legend Units Syntax Diagram

```
LegendUnits                ::= SEQUENCE {
     legend-unit                    [0] IMPLICIT Ratio,
     legend-unit-name               [1] IMPLICIT Text-String
                                    }
```

Figure E–61 illustrates the syntax used to create an angle construct.

Figure E–61 Angle Syntax Diagram

```
Angle              ::= FLOATING-POINT
```

Figure E–62 illustrates the syntax used to create an AngleRef construct.

Figure E–62 AngleRef Syntax Diagram

```
AngleRef               ::= CHOICE {
     angle-constant            [0] IMPLICIT Angle,
     angle-variable            [1] IMPLICIT VariableLabel
                               }
```

Figure E–63 illustrates the syntax used to create a measurement construct.

Figure E–63 Measurement Syntax Diagram

```
Measurement                ::= CHOICE {
     integer-constant          [0] IMPLICIT INTEGER,
     variable-measure          [1] IMPLICIT VariableLabel
                               }
```

Figure E–64 illustrates the syntax used to create a position construct.

Figure E–64 Position Syntax Diagram

```
Position                ::= SEQUENCE {
     x-coordinate                [0] XCoordinate,
     y-coordinate                [1] YCoordinate
                                 }
```

Figure E–65 illustrates the syntax used to create a ratio construct.

Figure E–65 Ratio Syntax Diagram

```
Ratio                       ::= SEQUENCE {
  numerator                       [0] IMPLICIT INTEGER DEFAULT 1,
  denominator                     [1] IMPLICIT INTEGER DEFAULT 100
                                  }
```

Figure E–66 illustrates the syntax used to create a right angle construct.

Figure E–66 Right Angle Syntax Diagram

```
RightAngle  ::= INTEGER { right(1),
                          left(2),
                          up(3),
                          down(4)
                        }
```

Figure E–67 illustrates the syntax used to create a size construct.

Figure E–67 Size Syntax Diagram

```
Size                        ::= Measurement
```

Figure E–68 illustrates the syntax used to create an *x*-coordinate construct.

Figure E–68 X-Coordinate Syntax Diagram

```
XCoordinate                ::= Measurement
```

Figure E–69 illustrates the syntax used to create a *y*-coordinate construct.

Figure E–69 Y-Coordinate Syntax Diagram

```
YCoordinate                ::= Measurement
```

Figure E–70 illustrates the syntax used to create a measurement units construct.

Figure E–70 Measurement Units Syntax Diagram

```
MeasurementUnits           ::= SEQUENCE {
    units-per-measurement        [0] IMPLICIT INTEGER,
    unit-name                    [1] IMPLICIT Text-String
                                 }
```

Figure E–71 illustrates the syntax used to create a named value construct.

Figure E–71 Named Value Syntax Diagram

```
NamedValue              ::= SEQUENCE {
   value-name                NamedValueTag,
   value-data                ValueData
                             }
```

Figure E–72 illustrates the syntax used to create a value data construct.

Figure E–72 Value Data Syntax Diagram

```
ValueData                ::= CHOICE {
     value-boolean                [0] IMPLICIT BOOLEAN,
     value-integer                [1] IMPLICIT INTEGER,
     value-text                   [2] IMPLICIT Text-String,
     value-general                [3] IMPLICIT OCTET STRING,
     value-reference              [4] IMPLICIT Reference,
     value-list                   [5] IMPLICIT SEQUENCE OF ValueData,
     value-external               [6] IMPLICIT EXTERNAL
                                  }
```

Figure E–73 illustrates the syntax used to create a named value list construct.

Figure E–73 Named Value List Syntax Diagram

```
NamedValueList ::= SEQUENCE OF NamedValue
```

Figure E–74 illustrates the syntax used to create a font number construct.

Figure E–74 Font Number Syntax Diagram

```
FontNumber               ::= INTEGER
```

Figure E–75 illustrates the syntax used to create a marker number construct.

Figure E–75 Marker Number Syntax Diagram

```
MarkerNumber             ::= INTEGER
```

Figure E–76 illustrates the syntax used to create a path number construct.

Figure E–76 Path Number Syntax Diagram

```
PathNumber                  ::= INTEGER
```

Figure E–77 illustrates the syntax used to create a pattern number construct.

Figure E–77 Pattern Number Syntax Diagram

```
PatternNumber               ::= INTEGER
```

Figure E–78 illustrates the syntax used to create a path definition construct.

Figure E–78 Path Definition Syntax Diagram

```
PathDefn                    ::= SEQUENCE  {
     path-number                  [0] IMPLICIT PathNumber,
     path-description             [1] IMPLICIT CompositePath,
     path-private                 [2] IMPLICIT NamedValueList          OPTIONAL
                                  }
```

Figure E–79 illustrates the syntax used to create a composite path construct.

Figure E–79 Composite Path Syntax Diagram

```
CompositePath               ::= SEQUENCE OF CHOICE {
     line-path-component          [0] IMPLICIT PolyLinePath,
     cubic-path-component         [1] IMPLICIT CubicBezierPath,
     arc-path-component           [2] IMPLICIT ArcPath,
     path-reference               [3] IMPLICIT PathNumber
                                  }
```

Figure E–80 illustrates the syntax used to create an arc path construct.

Figure E–80 Arc Path Syntax Diagram

```
ArcPath                    ::= SEQUENCE {
     arc-center-x                [0] XCoordinate,
     arc-center-y                [1] YCoordinate,
     arc-radius-x                [2] Size,
     arc-radius-delta-y          [3] Size DEFAULT { integer-constant 0 },
     arc-start                   [4] AngleRef
                                     DEFAULT { angle-constant 0.0 },
     arc-extent                  [5] AngleRef
                                     DEFAULT { angle-constant 360.0 },
     arc-rotation                [6] AngleRef
                                     DEFAULT { angle-constant 0.0 }
                               }
```

Figure E–81 illustrates the syntax used to create a cubic Bézier path construct.

Figure E–81 Cubic Bézier Path Syntax Diagram

```
CubicBezierPath            ::= SEQUENCE OF Measurement
```

Figure E–82 illustrates the syntax used to create a line definition construct.

Figure E–82 Line Definition Syntax Diagram

```
LineDefn              ::= SEQUENCE {
     line-style-number         [0] IMPLICIT LineStyleNumber,
     line-style-pattern        [1] IMPLICIT SEQUENCE OF INTEGER OPTIONAL,
     line-style-private        [2] IMPLICIT NamedValueList OPTIONAL
                               }
```

Figure E–83 illustrates the syntax used to create a polyline path construct.

Figure E–83 Polyline Path Syntax Diagram

```
PolyLinePath              ::= SEQUENCE OF Measurement
```

Figure E–84 illustrates the syntax used to create a pattern definition construct.

Figure E–84 Pattern Definition Syntax Diagram

```
PatternDefn               ::= SEQUENCE {
     pattern-number             [0] IMPLICIT PatternNumber,
     pattern-defn               CHOICE {
          solid-color                [1] Color,
          std-pattern                [2] IMPLICIT StandardPattern,
          raster-pattern             [3] IMPLICIT ImageDataUnit
                                     },
     pattern-private            [4] IMPLICIT NamedValueList OPTIONAL
                                 }
```

Figure E–85 illustrates the syntax used to create a standard pattern construct.

Figure E–85 Standard Pattern Syntax Diagram

```
StandardPattern           ::= SEQUENCE {
     std-pattern-number         [0] IMPLICIT INTEGER,
     pattern-colors             [1] IMPLICIT SEQUENCE OF PatternNumber
                                    DEFAULT {INTEGER 1, INTEGER 2}
                                }
```

Figure E–86 illustrates the syntax used to create a reference construct.

Figure E–86 Reference Syntax Diagram

```
Reference                  ::= SEQUENCE {
    ref-target                   [0] IMPLICIT SegmentLabel OPTIONAL,
    ref-x-index                  [1] IMPLICIT ExternalRefIndex OPTIONAL
                                 }
```

Figure E–87 illustrates the syntax used to create a segment attributes construct.

Figure E–87 Segment Attributes Syntax Diagram

```
SegmentAttributes          ::= SEQUENCE {
     private-attributes          [0] IMPLICIT NamedValueList          OPTIONAL,
     content-streams             [1] IMPLICIT SEQUENCE OF StreamTag   OPTIONAL,
     content-category            [2] IMPLICIT CategoryTag             OPTIONAL,
     segment-tags                [3] IMPLICIT SEQUENCE OF SegmentTag OPTIONAL,
     segment-bindings            [4] IMPLICIT SEQUENCE OF Binding     OPTIONAL,
     computed-content            [5] ComputeDefn                      OPTIONAL,
     structure-description       [6] StructureDefn                    OPTIONAL,
     language                    [7] IMPLICIT LanguageIndex           OPTIONAL,
     legend-units                [8] IMPLICIT LegendUnits             OPTIONAL,
     measurement-units           [9] IMPLICIT MeasurementUnits        OPTIONAL,
     alt-presentation           [10] IMPLICIT Text-String             OPTIONAL,
     layout-attributes          [11] TextLayout                       OPTIONAL,
     font-definitions           [12] IMPLICIT SEQUENCE OF FontDefn      OPTIONAL,
     pattern-definitions        [13] IMPLICIT SEQUENCE OF PatternDefn   OPTIONAL,
     path-definitions           [14] IMPLICIT SEQUENCE OF PathDefn      OPTIONAL,
     line-style-definitions     [15] IMPLICIT SEQUENCE OF LineDefn      OPTIONAL,
     content-defns              [16] IMPLICIT SEQUENCE OF ContentDefn   OPTIONAL,
     segment-type-defns         [17] IMPLICIT SEQUENCE OF SegTypeDefn   OPTIONAL,
     text-attributes            [18] IMPLICIT TextAttributes            OPTIONAL,
     line-attributes            [19] IMPLICIT LineAttributes            OPTIONAL,
     marker-attributes          [20] IMPLICIT MarkerAttributes          OPTIONAL,
     galley-attributes          [21] GalleyAttributes                   OPTIONAL, -- ANY
     image-attributes           [22] IMPLICIT ImageAttributes           OPTIONAL,
     frame-parameters           [23] IMPLICIT FrameParameters           OPTIONAL
                                }
```

Figure E–88 illustrates the syntax used to create a segment type definition construct.

Figure E–88 Segment Type Definition Syntax Diagram

```
SegTypeDefn                ::= SEQUENCE {
     type-label                  [0] IMPLICIT TypeDefnLabel,
     type-parent                 [1] IMPLICIT TypeDefnLabel     OPTIONAL,
     type-attributes             [2] IMPLICIT SegmentAttributes OPTIONAL,
     type-private                [3] IMPLICIT NamedValueList    OPTIONAL
                                 }
```

Figure E–89 illustrates the syntax used to create a structure definition construct.

Figure E–89 Structure Definition Syntax Diagram

```
StructureDefn           ::= CHOICE {
    sequence-structure          [0] IMPLICIT SEQUENCE OF OccurrenceDefn,
    set-structure               [1] IMPLICIT SEQUENCE OF OccurrenceDefn,
    choice-structure            [2] IMPLICIT SEQUENCE OF OccurrenceDefn
                                }
```

Figure E–90 illustrates the syntax used to create an occurrence definition construct.

Figure E–90 Occurrence Definition Syntax Diagram

```
OccurrenceDefn            ::= CHOICE {
    required-element              [0] StructureElement,
    optional-element              [1] StructureElement,
    repeat-element                [2] StructureElement,
    opt-repeat-element            [3] StructureElement
                                  }
```

Figure E–91 illustrates the syntax used to create a structure element construct.

Figure E–91 Structure Element Syntax Diagram

```
StructureElement            ::= CHOICE {
    expression-element              StructureDefn,
    referenced-type                 TypeDefnLabel
                                    }
```

Figure E–92 illustrates the syntax used to create a tag construct.

Figure E–92 Tag Syntax Diagram

```
Tag                     ::= ASCIIString
```

E.4 DDIF Syntax Diagrams

Figure E–93 illustrates the syntax used to create a category tag construct.

Figure E–93 Category Tag Syntax Diagram

```
CategoryTag                ::= Tag
```

Figure E–94 illustrates the syntax used to create a conformance tag construct.

Figure E–94 Conformance Tag Syntax Diagram

```
ConformanceTag             ::= Tag
```

Figure E–95 illustrates the syntax used to create a named value tag construct.

Figure E–95 Named Value Tag Syntax Diagram

```
NamedValueTag              ::= Tag
```

Figure E–96 illustrates the syntax used to create a segment tag construct.

Figure E–96 Segment Tag Syntax Diagram

```
SegmentTag                 ::= Tag
```

Figure E–97 illustrates the syntax used to create a storage system tag construct.

Figure E–97 Storage System Tag Syntax Diagram

```
StorageSystemTag           ::= Tag
```

Figure E–98 illustrates the syntax used to create a stream tag construct.

Figure E–98 Stream Tag Syntax Diagram

```
StreamTag                ::= Tag
```

Figure E–99 illustrates the syntax used to create a transformation construct.

Figure E–99 Transformation Syntax Diagram

```
Transformation              ::= SEQUENCE OF CHOICE {
     x-scale                        [0] IMPLICIT FLOATING-POINT,
     y-scale                        [1] IMPLICIT FLOATING-POINT,
     x-translation                  [2] IMPLICIT FLOATING-POINT,
     y-translation                  [3] IMPLICIT FLOATING-POINT,
     xy-rotate                      [4] IMPLICIT Angle,
     xy-skew                        [5] IMPLICIT Angle,
     transform-2x3                  [6] IMPLICIT SEQUENCE OF FLOATING-POINT,
     transform-3x3                  [7] IMPLICIT SEQUENCE OF FLOATING-POINT
                                    }
```

Figure E–100 illustrates the syntax used to create a variable binding construct.

Figure E–100 Variable Binding Syntax Diagram

```
Binding                      ::= SEQUENCE {
     variable-name                   [0] IMPLICIT VariableLabel,
     variable-value                  CHOICE {
          counter-variable                [1] IMPLICIT CounterDefn,
          computed-variable               [2] IMPLICIT StringExpression,
          list-variable                   [3] IMPLICIT RecordList
                                          }
                                     }
```

Figure E–101 illustrates the syntax used to create a counter definition construct.

Figure E–101 Counter Definition Syntax Diagram

```
CounterDefn                ::= SEQUENCE {
     counter-trigger             CHOICE {
          counts-tagged-segments      [0] IMPLICIT SegmentTag,
          counts-layout-objs          [1] IMPLICIT LayoutObjectType
                                      } OPTIONAL,
     counter-init               [2] Expression DEFAULT { exp-integer 1 },
     counter-style              [3] IMPLICIT SEQUENCE OF CounterStyle OPTIONAL,
     counter-type               [4] IMPLICIT INTEGER {
          military(1),
          office(2),
          page-relative(3) }         DEFAULT office
                                 }
```

Figure E–102 illustrates the syntax used to create a layout object type construct.

Figure E–102 Layout Object Type Syntax Diagram

```
LayoutObjectType           ::= INTEGER { document-layout-object(1),
                                         page-set-layout-object(2),
                                         page-layout-object(3),
                                         frame-layout-object(4),
                                         block-layout-object(5),
                                         line-layout-object(6)
                                         }
```

Figure E–103 illustrates the syntax used to create an expression construct.

Figure E–103 Expression Syntax Diagram

```
Expression                 ::= CHOICE {
     exp-integer                   [0] IMPLICIT INTEGER,
     exp-variable                  [1] IMPLICIT VariableLabel
                                   }
```

Figure E–104 illustrates the syntax used to create a counter style construct.

Figure E–104 Counter Style Syntax Diagram

```
CounterStyle              ::= CHOICE {
     number-style                  [0] IMPLICIT INTEGER {
          arabic(1),    l-roman(2),
          u-roman(3),   l-latin(4),
          u-latin(5),   w-arabic(6),
          wl-roman(7),  wu-roman(8),
          wl-latin(9),  wu-latin(10),
          w-katakana-50(11),
          w-katakana-iroha(12),
          hebrew(13)                                  },
     bullet-style               [1] IMPLICIT SEQUENCE OF Character-String,
     style-separator            [2] IMPLICIT Character-String
                                }
```

Figure E–105 illustrates the syntax used to create a string expression construct.

Figure E–105 String Expression Syntax Diagram

```
StringExpression          ::= SEQUENCE OF CHOICE {
     text-element                  [0] IMPLICIT Character-String,
     variable-ref-element          [1] IMPLICIT VariableLabel
                                   }
```

Figure E–106 illustrates the syntax used to create a record list construct.

Figure E–106 Record List Syntax Diagram

```
RecordList ::= SEQUENCE OF RecordDefn
```

Figure E–107 illustrates the syntax used to create a record definition construct.

Figure E–107 Record Definition Syntax Diagram

```
RecordDefn              ::= SEQUENCE {
     record-type              [0] IMPLICIT TypeDefnLabel,
     record-tag               [1] IMPLICIT SegmentTag,
     record-contents          [2] IMPLICIT SEQUENCE OF VariableLabel
                              }
```

Figure E–108 illustrates the syntax used to create a generic layout construct.

Figure E–108 Generic Layout Syntax Diagram

```
GenericLayout             ::= [APPLICATION 31] IMPLICIT SEQUENCE {
     gl-private-data            [0] IMPLICIT NamedValueList OPTIONAL,
     gl-page-descriptions       [1] IMPLICIT SEQUENCE OF PageDescription
                                }
```

Figure E–109 illustrates the syntax used to create a page description construct.

Figure E–109 Page Description Syntax Diagram

```
PageDescription           ::= SEQUENCE {
     pd-label                   [0] IMPLICIT PageDescLabel,
     pd-private-data            [1] IMPLICIT NamedValueList OPTIONAL,
     pd-desc                    CHOICE {
          page-set-desc              [2] IMPLICIT PageSet,
          page-layout                [3] IMPLICIT PageLayout
                                     }
                                }
```

Figure E–110 illustrates the syntax used to create a page set construct.

Figure E–110 Page Set Syntax Diagram

```
PageSet                  ::= SEQUENCE OF PageSelect

PageSelect               ::= SEQUENCE {
     page-side-criteria        [0] IMPLICIT INTEGER {
          left-page(1),
          right-page(2),
          either-page(3)                          } DEFAULT either-page,
     selected-page-layout   CHOICE {
          select-by-label     [1] IMPLICIT PageLayoutLabel,
          select-by-defn      [2] IMPLICIT PageLayout
                            }
                              }
```

Figure E–111 illustrates the syntax used to create a page layout construct.

Figure E–111 Page Layout Syntax Diagram

```
PageLayout               ::= SEQUENCE {
     page-layout-id             [0] IMPLICIT PageLayoutLabel,
     page-size                  [1] IMPLICIT GenSize,
     page-orientation           [2] IMPLICIT INTEGER {
           portrait(1),
           landscape(2) }           DEFAULT  portrait,
     page-prototype             [3] IMPLICIT PageLayoutLabel OPTIONAL,
     page-content               [4] IMPLICIT PageFrame       OPTIONAL
                                }

PageFrame              ::= Content -- Must be a frame
```

Figure E–112 illustrates the syntax used to create a layout primitive construct.

Figure E–112 Layout Primitive Syntax Diagram

```
LayoutPrimitive          ::= [APPLICATION 35] ANY
```

Figure E–113 illustrates the syntax used to create a layout galley construct.

Figure E–113 Layout Galley Syntax Diagram

```
LayoutGalley              ::= [APPLICATION 36] IMPLICIT SEQUENCE {
     galley-id                 [0] IMPLICIT GalleyLabel,
     galley-bounding-box       [1] IMPLICIT BoundingBox,
     galley-outline            [2] IMPLICIT CompositePath          OPTIONAL,
     galley-flags              [3] IMPLICIT BIT STRING {
          galley-vertical-align(0),
          galley-border(1),
          galley-autoconnect(2),
          galley-background-fill(3)
          }                                                        OPTIONAL,
     galley-streams           [4] IMPLICIT SEQUENCE OF StreamTag OPTIONAL,
     galley-successor         CHOICE {
          generic-galley           [5] IMPLICIT GalleyLabel,
          specific-galley          [6] IMPLICIT GalleyLabel,
          no-successor-galley      [7] IMPLICIT NULL
                                   }
                                }
```

Figure E–114 illustrates the syntax used to create a galley attributes construct.

Figure E–114 Galley Attributes Syntax Diagram

```
GalleyAttributes          ::= [APPLICATION 37] IMPLICIT SEQUENCE {
     galley-top-margin         [0] Measurement                    OPTIONAL,
     galley-left-margin        [1] Measurement                    OPTIONAL,
     galley-right-margin       [2] Measurement                    OPTIONAL,
     galley-bottom-margin      [3] Measurement                    OPTIONAL
                               }
```

Figure E–115 illustrates the syntax used to create a specific layout construct.

Figure E–115 Specific Layout Syntax Diagram

```
SpecificLayout        ::= [APPLICATION 32] IMPLICIT SEQUENCE OF CHOICE {
     specific-page          [0] IMPLICIT PageDescription,
     referenced-page        [1] IMPLICIT PageDescLabel
                            }
```

Figure E–116 illustrates the syntax used to create a wrap attributes construct.

Figure E–116 Wrap Attributes Syntax Diagram

```
WrapAttributes              ::= [APPLICATION 33] IMPLICIT SEQUENCE {
     wrap-format                 [0] IMPLICIT Format OPTIONAL,
     quad-format                 [1] IMPLICIT Format OPTIONAL,
     hyphenation-flags           [2] IMPLICIT BIT STRING {
          hyphenation-allowed(0),
          paragraph-end(1),
          galley-end(2),
          page-end(3),
          capitalized-word(4) }      OPTIONAL,
     maximum-hyph-lines          [3] IMPLICIT INTEGER OPTIONAL,
     maximum-orphan-size         [4] IMPLICIT INTEGER OPTIONAL,
     maximum-widow-size          [5] IMPLICIT INTEGER OPTIONAL
                                 }
```

Figure E–117 illustrates the syntax used to create a layout attributes construct.

Figure E–117 Layout Attributes Syntax Diagram

```
LayoutAttributes          ::= [APPLICATION 34] IMPLICIT SEQUENCE {
     initial-directive          [0] IMPLICIT Directive     OPTIONAL,
     galley-select              [1] IMPLICIT GalleyLabel   OPTIONAL,
     break-before               [2] IMPLICIT BreakCriteria OPTIONAL,
     break-within               [3] IMPLICIT BreakCriteria OPTIONAL,
     break-after                [4] IMPLICIT BreakCriteria OPTIONAL,
     initial-indent             [5] Measurement OPTIONAL,
     left-indent                [6] Measurement OPTIONAL,
     right-indent               [7] Measurement OPTIONAL,
     space-before               [8] Measurement OPTIONAL,
     space-after                [9] Measurement OPTIONAL,
     leading                   [10] IMPLICIT Escapement OPTIONAL,
     tab-stops                 [11] IMPLICIT TabStopList OPTIONAL
                                }
```

Figure E–118 illustrates the syntax used to create a break criteria construct.

Figure E–118 Break Criteria Syntax Diagram

```
BreakCriteria              ::= INTEGER {
     break-always(1),
     break-never(2),
     break-if-needed(3)                }
```

Figure E–119 illustrates the syntax used to create a general measure construct.

Figure E–119 General Measure Syntax Diagram

```
GenMeasure                 ::= SEQUENCE {
     nominal-measure             [0] Measurement DEFAULT { integer-constant 0 },
     stretch-measure             [1] Measurement DEFAULT { integer-constant 0 },
     shrink-measure              [2] Measurement DEFAULT { integer-constant 0 }
                                 }
```

Figure E–120 illustrates the syntax used to create a general size construct.

Figure E–120 General Size Syntax Diagram

```
GenSize                    ::= SEQUENCE {
     x-size                      [0] IMPLICIT GenMeasure,
     y-size                      [1] IMPLICIT GenMeasure
                                 }
```

Figure E–121 illustrates the syntax used to create a tab stop list construct.

Figure E–121 Tab Stop List Syntax Diagram

```
TabStopList                ::= SEQUENCE OF TabStop
```

Figure E–122 illustrates the syntax used to create a tab stop construct.

Figure E–122 Tab Stop Syntax Diagram

```
TabStop                    ::= SEQUENCE {
     horizontal-position         [0] Measurement,
     tab-stop-type               [1] IMPLICIT INTEGER {
          left-tab(1),
          center-tab(2),
          right-tab(3),
          decimal-tab(4) }            DEFAULT left-tab,
     tab-stop-leader             [2] IMPLICIT Character-String OPTIONAL
                                 }
```

F DDIF Fill Patterns

This appendix describes the various fill patterns supported by the CDA Toolkit. These fill patterns correspond to those used by the Graphics Kernel System (GKS). They are valid for the following aggregate items:

- The text mask pattern item (DDIF$_SGA_TXT_MASK_PATTERN) in the DDIF$_SGA aggregate
- The line mask pattern item (DDIF$_SGA_LIN_MASK_PATTERN) in the DDIF$_SGA aggregate
- The line interior pattern item (DDIF$_SGA_LIN_INTERIOR_ PATTERN) in the DDIF$_SGA aggregate
- The marker mask pattern item (DDIF$_SGA_MKR_MASK_PATTERN) in the DDIF$_SGA aggregate
- The pattern number item (DDIF$_PTD_NUMBER) in the DDIF$_PTD aggregate
- The pattern colors item (DDIF$_PTD_PAT_NUMBER) in the DDIF$_ PTD aggregate

Table F–1 describes each predefined fill pattern, its symbolic name, and its corresponding DDIF pattern number. Figure F–1 illustrates each predefined fill pattern.

Table F–1 DDIF Fill Patterns

Pattern Name	Number	Description
DDIF$K_PATT_BACKGROUND	1	The pattern is white.
DDIF$K_PATT_FOREGROUND	2	The pattern is black.
DDIF$K_PATT_VERT1_1	3	The ratio of black to white vertical lines in the pattern is 1:1.
DDIF$K_PATT_VERT1_3	4	The ratio of black to white vertical lines in the pattern is 1:3.
DDIF$K_PATT_VERT2_2	5	The ratio of black to white vertical lines in the pattern is 2:2.
DDIF$K_PATT_VERT3_1	6	The ratio of black to white vertical lines in the pattern is 3:1.
DDIF$K_PATT_VERT1_7	7	The ratio of black to white vertical lines in the pattern is 1:7.
DDIF$K_PATT_VERT2_6	8	The ratio of black to white vertical lines in the pattern is 2:6.

(continued on next page)

Table F–1 (Cont.) DDIF Fill Patterns

Pattern Name	Number	Description
DDIF$K_PATT_VERT4_4	9	The ratio of black to white vertical lines in the pattern is 4:4.
DDIF$K_PATT_VERT6_2	10	The ratio of black to white vertical lines in the pattern is 6:2.
DDIF$K_PATT_HORIZ1_1	11	The ratio of black to white horizontal lines in the pattern is 1:1.
DDIF$K_PATT_HORIZ1_3	12	The ratio of black to white horizontal lines in the pattern is 1:3.
DDIF$K_PATT_HORIZ2_2	13	The ratio of black to white horizontal lines in the pattern is 2:2.
DDIF$K_PATT_HORIZ3_1	14	The ratio of black to white horizontal lines in the pattern is 3:1.
DDIF$K_PATT_HORIZ1_7	15	The ratio of black to white horizontal lines in the pattern is 1:7.
DDIF$K_PATT_HORIZ2_6	16	The ratio of black to white horizontal lines in the pattern is 2:6.
DDIF$K_PATT_HORIZ4_4	17	The ratio of black to white horizontal lines in the pattern is 4:4.
DDIF$K_PATT_HORIZ6_2	18	The ratio of black to white horizontal lines in the pattern is 6:2.
DDIF$K_PATT_GRID4	19	The area enclosed in each grid box is 4 units.
DDIF$K_PATT_GRID8	20	The area enclosed in each grid box is 8 units.
DDIF$K_PATT_UPDIAG1_3	21	The ratio of black to white upward diagonal lines (going up from left to right) in the pattern is 1:3.
DDIF$K_PATT_UPDIAG2_2	22	The ratio of black to white upward diagonal lines (going up from left to right) in the pattern is 2:2.
DDIF$K_PATT_UPDIAG3_1	23	The ratio of black to white upward diagonal lines (going up from left to right) in the pattern is 3:1.
DDIF$K_PATT_UPDIAG1_7	24	The ratio of black to white upward diagonal lines (going up from left to right) in the pattern is 1:7.

(continued on next page)

Table F–1 (Cont.) DDIF Fill Patterns

Pattern Name	Number	Description
DDIF$K_PATT_UPDIAG2_6	25	The ratio of black to white upward diagonal lines (going up from left to right) in the pattern is 2:6.
DDIF$K_PATT_UPDIAG4_4	26	The ratio of black to white upward diagonal lines (going up from left to right) in the pattern is 4:4.
DDIF$K_PATT_UPDIAG6_2	27	The ratio of black to white upward diagonal lines (going up from left to right) in the pattern is 6:2.
DDIF$K_PATT_DOWNDIAG1_3	28	The ratio of black to white downward diagonal lines (going down from left to right) in the pattern is 1:3.
DDIF$K_PATT_DOWNDIAG2_2	29	The ratio of black to white downward diagonal lines (going down from left to right) in the pattern is 2:2.
DDIF$K_PATT_DOWNDIAG3_1	30	The ratio of black to white downward diagonal lines (going down from left to right) in the pattern is 3:1.
DDIF$K_PATT_DOWNDIAG1_7	31	The ratio of black to white downward diagonal lines (going down from left to right) in the pattern is 1:7.
DDIF$K_PATT_DOWNDIAG2_6	32	The ratio of black to white downward diagonal lines (going down from left to right) in the pattern is 2:6.
DDIF$K_PATT_DOWNDIAG4_4	33	The ratio of black to white downward diagonal lines (going down from left to right) in the pattern is 4:4.
DDIF$K_PATT_DOWNDIAG6_2	34	The ratio of black to white downward diagonal lines (going down from left to right) in the pattern is 6:2.
DDIF$K_PATT_BRICK_HORIZ	35	The pattern is composed of bricks oriented in a horizontal direction.
DDIF$K_PATT_BRICK_VERT	36	The pattern is composed of bricks oriented in a vertical direction.
DDIF$K_PATT_BRICK_DOWNDIAG	37	The pattern is composed of bricks oriented in a downward diagonal pattern (going down from left to right).

(continued on next page)

Table F–1 (Cont.) DDIF Fill Patterns

Pattern Name	Number	Description
DDIF$K_PATT_BRICK_UPDIAG	38	The pattern is composed of bricks oriented in an upward diagonal pattern (going up from left to right).
DDIF$K_PATT_GREY4_16D	39	The ratio of black to white dots in the pattern is 4:16.
DDIF$K_PATT_GREY12_16D	40	The ratio of black to white dots in the pattern is 12:16.
DDIF$K_PATT_BASKET_WEAVE	41	The pattern is composed of a basket-weave pattern.
DDIF$K_PATT_SCALE_DOWN	42	The pattern is composed of downward-oriented scales.
DDIF$K_PATT_SCALE_UP	43	The pattern is composed of upward-oriented scales.
DDIF$K_PATT_SCALE_RIGHT	44	The pattern is composed of rightward-oriented scales.
DDIF$K_PATT_SCALE_LEFT	45	The pattern is composed of leftward-oriented scales.
DDIF$K_PATT_FILLER6	46	The pattern is a filler pattern.
DDIF$K_PATT_FILLER7	47	The pattern is a filler pattern.
DDIF$K_PATT_GREY1_16	48	The ratio of black to white dots in the pattern is 1:16.
DDIF$K_PATT_GREY2_16	49	The ratio of black to white dots in the pattern is 2:16.
DDIF$K_PATT_GREY3_16	50	The ratio of black to white dots in the pattern is 3:16.
DDIF$K_PATT_GREY4_16	51	The ratio of black to white dots in the pattern is 4:16.
DDIF$K_PATT_GREY5_16	52	The ratio of black to white dots in the pattern is 5:16.
DDIF$K_PATT_GREY6_16	53	The ratio of black to white dots in the pattern is 6:16.
DDIF$K_PATT_GREY7_16	54	The ratio of black to white dots in the pattern is 7:16.
DDIF$K_PATT_GREY8_16	55	The ratio of black to white dots in the pattern is 8:16.
DDIF$K_PATT_GREY9_16	56	The ratio of black to white dots in the pattern is 9:16.
DDIF$K_PATT_GREY10_16	57	The ratio of black to white dots in the pattern is 10:16.
DDIF$K_PATT_GREY11_16	58	The ratio of black to white dots in the pattern is 11:16.

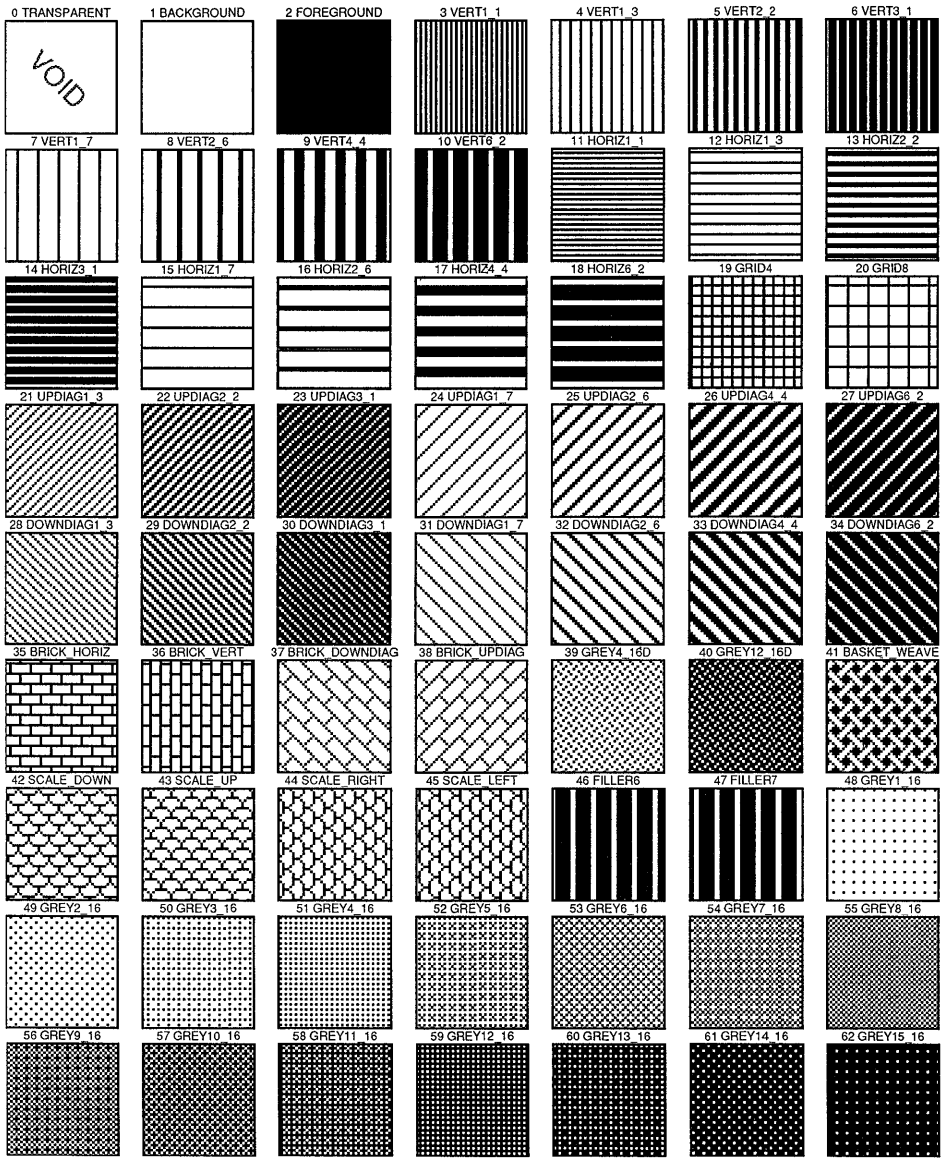
(continued on next page)

Table F–1 (Cont.) DDIF Fill Patterns

Pattern Name	Number	Description
DDIF$K_PATT_GREY12_16	59	The ratio of black to white dots in the pattern is 12:16.
DDIF$K_PATT_GREY13_16	60	The ratio of black to white dots in the pattern is 13:16.
DDIF$K_PATT_GREY14_16	61	The ratio of black to white dots in the pattern is 14:16.
DDIF$K_PATT_GREY15_16	62	The ratio of black to white dots in the pattern is 15:16.

DDIF Fill Patterns

Figure F–1 CDA Fill Patterns



Glossary of Terms

aggregate: An in-memory structure that is used to pass compound document data between the application and the CDA Toolkit routines. An aggregate corresponds to a manageable unit of the compound document. Aggregates are typed and self-describing; the type of an aggregate is indicated by a symbolic constant. An aggregate can be a member of an aggregate sequence, which can be traversed from beginning to end. Aggregates are defined for such objects as document roots, document descriptors, document headers, document segments, text content, and so on.

AngleRef enumeration: A compound document data type that is an enumeration specifying the data type of an item of DDIF type AngleRef, which is encoded as a floating-point or string value.

attribute: A term used to describe content characteristics such as font, line thickness, and color.

binary relative time: A compound document data type that specifies a binary relative time.

bit string: A compound document data type that is encoded as a string of bits. The length of the item is expressed in bits.

Boolean: A compound document data type, encoded as a byte, that represents a Boolean value.

byte: A compound document data type that is encoded as a byte.

character string: A compound document data type that is encoded as a string of bytes in a particular character set.

compound document: A unified collection of data that can be edited, formatted, or otherwise processed as a document.

computed content: Document content (most often text content) that is calculated based on the current formatting state or other inclusion of external data.

content: The class of data that makes up the fundamental units of documents. Document content includes characters, lines, raster images, and so on.

content reference: A shorthand notation for the phrase "reference to generic content." A content reference is a relationship in a revisable document that defines the situation in which a content reference causes the generic content to be inserted into the final form when the document is formatted.

document: An entire hierarchical structure in memory, created by the CDA Toolkit routines.

document content: See *content*.

document segment: See *segment*.

enumeration: A compound document data type that is encoded as a longword integer. An item that is encoded as an enumeration must specify the possible values for the enumeration. If the item following the enumeration item is of DDIF type variable, then the value selected for the enumeration item affects the encoding of the subsequent item.

expression enumeration: A compound document data type that is an enumeration that specifies the data type of an item of DDIF type expression.

final form: Stage of a document in which all the formatting decisions (such as hyphenation, line breaks, and page breaks) have been resolved.

formatting: The process of fixing text in galleys. Formatting involves breaking the stream of characters and floating frames into lines that fit within the assigned galleys. Formatting can also involve optimization of page layouts, the selection of appropriate page templates, and hyphenation decisions.

galley: A rectangular guide, such as a column or footnote area.

generic attributes: A relationship in a revisable document that defines attributes that can be applied to a number of segments, as opposed to being associated with a single segment.

generic content: A relationship in a revisable document that defines document content that can be included in multiple places in the document.

generic type: A relationship in a revisable document that defines a set of attributes and processing tags that define a type. Elements of the document can reference a defined type and become an instance of the type, thus inheriting the attributes and processing characteristics of the generic type.

graphics content: Content that consists of primitives such as polylines and filled areas.

handle: The identifier of an aggregate, stream, file, front end, or back end.

hard content: Content that is entered by the creator of the document.

image content: Content that consists of digitized images represented by actual values of monochrome, grey-level, or color images.

inheritance: A relationship in a revisable document that defines a method for defaulting the attributes of content so that each segment of content does not need to specify all of its attributes. Instead, each segment inherits the attributes of the surrounding segment, and specifies only the differences between the attributes of its content and those of the surrounding content.

integer: A compound document data type that represents a longword integer.

item: A specific unit of information stored in an aggregate.

item change list: A compound document data type that specifies a vector of longwords in which each longword contains the item code of an item in a segment attributes aggregate that has changed.

kerning: In typesetting, subtracting the space between two characters so that they appear closer together.

leaders: In composition, rows of dashes or dots that are used to guide the eye across the page. Leaders are used in tabular work, programs, tables of contents, and so on.

leading: In composition, the distance between lines of type measured in points.

longword: A compound document data type representing a longword bit-encoded structure. The bits are interpreted according to a defined structure.

measurement enumeration: A compound document data type that is an enumeration specifying the data type of an item of DDIF type measurement, which is encoded as an integer or string.

object identifier: A compound document data type that contains two or more longwords that specify the value of the DDIS type object identifier.

raster image content: See *image content*.

revisable document: A document that contains abstract relationships between the components of the document. That is, the characteristics of the document that determine the final appearance are specified as parameters and directives that are used to create the final form.

root aggregate: An aggregate that represents the root of the in-memory document hierarchy. A root aggregate contains context private to the Toolkit routines. The type of a root aggregate is DDIF$_DDF.

root segment: A top level segment that contains the document content. This document content can consist of content aggregates as well as nested segments. If a document contains only one segment, that segment is the root segment and contains all of the document content. If the document contains multiple segments, they must be nested within a root segment.

segment: A quantity of content that is set off from surrounding data by a change in presentation or processing attributes.

sequence: A linked series of aggregates.

single precision floating: A compound document data type that specifies a VMS F_floating point value.

soft content: Content that is generated by software and is subject to recalculation when the document is revised.

specific attributes: Attributes that are associated only with a single segment of content. These types of attributes are deliberately limited to a specific segment of the document.

stream: An access path by which encoded compound document data is read from or written to a storage medium.

string: A compound document data type that is encoded as a string of bytes. The length of the string is also specified in bytes.

string with add-info: A compound document data type, encoded as a string of bytes, that represents the value of the DDIF type tag.

style guide: A relationship in a revisable document that defines a collection of generic types defined for use by a set of documents.

text content: Content that consists of text in ASCII and alternate character sets.

type reference: A shorthand notation for the phrase "reference to generic type." A type reference is a relationship in a revisable document that defines the situation of segments referencing the same generic type and therefore inheriting common attributes and processing and presentation styles.

variable: A compound document data type for which the data type of the item is determined by a preceding enumeration item. The enumeration item determines the data type of the variable item.

variables: A relationship in a revisable document that defines content that can be generated based on the values of variables, thereby ensuring that multiple elements of content are identical, have the same position, or can be modified by standard functions.

word: A compound document data type that is encoded as a word.

Index

A

Aggregate
  See also Root aggregate
  copying • 4–5, CDA–33
  creating • 4–4, CDA–35
  definition of • 4–1
  deleting • 4–5, CDA–54
  determining number of array elements in • CDA–80
  handle of • 4–1
  inserting • CDA–91
  inserting into a sequence • 4–5
  locating an item in • CDA–96
  locating in a sequence • 4–5
  locating next in sequence • CDA–99
  populating • 4–4
  reading from a front end • CDA–26
  reading from a stream • CDA–77
  reading the next from a stream • 4–11
  removing from a document • 4–5, CDA–119
  removing from a sequence • 4–5, CDA–130
  writing the contents of • 4–11, CDA–123, CDA–131
Aggregate-method conversion • 5–10 to 5–13
  Close • 5–13
  Get-Aggregate • 5–12
  Get-Position • 5–12
Aggregate transfer • 4–10 to 4–13
AGGREGATE TYPE TO OBJECT ID routine • 4–4, CDA–3
Allocation routine • CDA–40, CDA–44, CDA–48, CDA–110, CDA–113
Alternate presentation • 6–41
Analysis back end • 2–15
AngleRef enumeration • 4–7
Arc
  controlling the rendition of • 6–18
  specifying angle of rotation of • 6–19
  specifying center *x*-coordinate of • 6–18
  specifying center *y*-coordinate of • 6–18
  specifying delta *y* radius of • 6–19
  specifying the extent of • 6–19
  specifying the starting angle of • 6–19
  specifying *x* radius of • 6–18
Arc content aggregate • 6–18 to 6–20
Arc content aggregate (cont'd.)
  See also DDIF$_ARC aggregate
  arc extent indicator item in • 6–19
  arc start indicator item in • 6–19
  center *x* indicator item in • 6–18
  center *y* indicator item in • 6–18
  delta *y* indicator item • 6–19
  flags item in • 6–18
  items in • 6–19t, D–4t
  rotation indicator item in • 6–19
  x radius item in • 6–18
Attribute
  definition of • 3–1, 4–1
  for layout • 3–16
  for wrapping text • 3–16
  generic • 3–6
  inheritance of • 3–8
  layout • 6–9
  precedence of • 3–9
  specific • 3–7
  specifying for a galley • 6–55
  specifying for a line • 6–52 to 6–54
  specifying for a marker • 6–55
  specifying for an image • 6–56 to 6–58
  specifying for document content • 6–9
  specifying for image component space • 6–58 to 6–60
  specifying for text • 6–48 to 6–52
  text • 6–9

B

Back end • 5–14 to 5–18, CDA–19
  analysis • 2–15
  DDIF • 2–10 to 2–11
  DDIF$WRITE_*format* entry point • 5–14
  entry point • CDA–11
  PostScript • 2–11 to 2–15
  text • 2–11
BMU (Basic Measurement Unit)
  definition of • 6–40
Buffer
  specifyng size of • 5–13
Bézier curve
  controlling rendition of • 6–15
  specifying layout of • 6–16

Bézier curve aggregate • 6–15 to 6–16
See also DDIF$_BEZ aggregate
curve path indicator item in • 6–16
flags item in • 6–15
items in • 6–16t, D–3t

C

CDA$AGGREGATE_TYPE_TO_OBJECT_ID • 4–4, CDA–3
CDA$CLOSE_FILE • 4–2, CDA–5
CDA$CLOSE_STREAM • 4–3, CDA–7
CDA$CLOSE_TEXT_FILE • 4–3, CDA–8
CDA$CONVERT • 2–3, 2–4 to 2–7, CDA–9
CDA$CONVERT_AGGREGATE • 4–10, CDA–26
CDA$CONVERT_DOCUMENT • 4–10, CDA–29
CDA$CONVERT_POSITION • 4–11, CDA–31
CDA$COPY_AGGREGATE • 4–5, CDA–33
CDA$CREATE_AGGREGATE • 4–4, CDA–35
CDA$CREATE_FILE • 4–2, CDA–37
CDA$CREATE_ROOT_AGGREGATE • 4–4, CDA–42
CDA$CREATE_STREAM • 4–3, CDA–46
CDA$CREATE_TEXT_FILE • 4–3, CDA–51
CDA$DELETE_AGGREGATE • 4–5, CDA–54
CDA$DELETE_ROOT_AGGREGATE • 4–4, CDA–56
CDA$ENTER_SCOPE • 4–12, CDA–57
CDA$ERASE_ITEM • 4–9, CDA–68
CDA$FIND_DEFINITION • CDA–70
CDA$FIND_TRANSFORMATION • CDA–73
CDA$FLUSH_STREAM • 4–3, CDA–75
CDA$GET_AGGREGATE • 4–11, CDA–77
CDA$GET_ARRAY_SIZE • 4–9, CDA–80
CDA$GET_DOCUMENT • 4–10, CDA–82
CDA$GET_EXTERNAL_ENCODING • CDA–84
CDA$GET_STREAM_POSITION • CDA–86
CDA$GET_TEXT_POSITION • CDA–89
CDA$INSERT_AGGREGATE • 4–5, CDA–91
CDA$LEAVE_SCOPE • 4–12, CDA–94
CDA$LOCATE_ITEM • 4–9, CDA–96
CDA$NEXT_AGGREGATE • 4–5, CDA–99
CDA$OBJECT_ID_TO_AGGREGATE_TYPE • 4–4, CDA–101
CDA$OPEN_CONVERTER • 4–13, CDA–103
CDA$OPEN_FILE • 4–2, CDA–106
CDA$OPEN_STREAM • 4–3, CDA–112
CDA$OPEN_TEXT_FILE • 4–3, CDA–116
CDA$PRUNE_AGGREGATE • 4–5, CDA–119
CDA$PRUNE_POSITION • 4–10, CDA–121
CDA$PUT_AGGREGATE • 4–11, CDA–123
CDA$PUT_DOCUMENT • 4–10, CDA–126
CDA$READ_TEXT_FILE • 4–3, CDA–128
CDA$REMOVE_AGGREGATE • 4–5, CDA–130
CDA$STORE_ITEM • 4–9, CDA–131
CDA$WRITE_TEXT_FILE • 4–3, CDA–137
CDA converter kernel • 2–2
CDA Toolkit • 1–3
Character set
identifiers for • 6–10t
CLOSE FILE routine • 4–2, CDA–5
CLOSE STREAM routine • 4–3, CDA–7
CLOSE TEXT FILE routine • 4–3, CDA–8
Color
See Pattern definition
Composite path
arc component of • 6–32
Bézier component of • 6–32
indicating angle of rotation of • 6–34
indicating center *x*-coordinate of an arc in • 6–33
indicating center *y*-coordinate of an arc in • 6–33
indicating delta *y*-radius of an arc in • 6–33
indicating extent of an arc in • 6–33
indicating type of path defined in • 6–32
indicating *x* radius of an arc in • 6–33
polyline component of • 6–32
referencing a component of • 6–32, 6–34
specifying layout of the curve in • 6–32
specifying polyline path in • 6–32
specifying starting angle of an arc in • 6–33
Composite path aggregate • 6–32 to 6–34
See also DDIF$_PTH aggregate
arc center *x* indicator item in • 6–33
arc center *y* indicator item in • 6–33
arc extent indicator item in • 6–33
arc radius delta *y* indicator item in • 6–33
arc radius *x* indicator item in • 6–33
arc rotation indicator item in • 6–34
arc start indicator item in • 6–33
curve path indicator item in • 6–32
items in • 6–34t, D–7t
line path indicator item in • 6–32
path indicator item in • 6–32
path reference item in • 6–34
Compound document • 1–2
validating contents of • CDA–106
viewing • 2–7
Computed content • 3–1, 3–5
copied • 6–38
cross-reference • 6–38
function • 6–38

Computed content (cont'd.)
indicating function parameters for • 6–39
specifying an index into a list of references for • 6–38
specifying attributes for • 6–37 to 6–39
specifying function name in • 6–38
specifying label of a segment being referenced by • 6–38
specifying label of the target segment • 6–38
specifying name of the variable in • 6–38
specifying reference index for • 6–38
specifying the name of the referenced variable in • 6–38
variable • 6–38
Content • 3–1 to 3–14
See also Root segment
computed • 3–1, 3–5
specifying attributes for • 6–37 to 6–39
definition of • 3–1
generic • 3–8 to 3–9
referencing • 3–9
graphics • 3–1, 3–5
hard • 3–1
image • 3–1, 3–5, 6–21 to 6–22
indicating relational position of a segment of • 6–46
indicating the processing characteristics of • 6–9
normal alignments for character orientations in • 6–45t
presentation attributes of • 6–9
presentation styles for • 6–6
restricted • 3–6, 6–22 to 6–25
external • 6–22 to 6–23
private • 6–24 to 6–25
separating from layout • 1–7
soft • 3–1
specifying alternate presentation string for • 6–41
specifying character horizontal alignment point for • 6–44
specifying character vertical alignment point for • 6–44
specifying general character set for • 6–10
specifying general layout attributes for • 6–42
specifying Latin1 character set for • 6–10
specifying magnitude of coordinate system ratio for • 6–40
specifying name of the measurement system of • 6–40
specifying number of units per inch of • 6–40
specifying precision in coordinate system ratio for • 6–40
specifying string format of • 6–42
Content (cont'd.)
specifying the character orientation in • 6–43
specifying the layout of • 6–41
specifying the name of the coordinate system of • 6–40
specifying the string imaging path of • 6–42
specifying world coordinate system for • 6–40
specifying wrap attributes for • 6–42
text • 3–1, 3–5, 6–9
general • 6–10
Latin1 • 6–10
Content attribute • 3–8
Content category • 6–8
Content category tags
text content • 6–9
Content definition
specifying elements in • 6–70
specifying index into list of external references in • 6–70
specifying label of • 6–69
specifying label of the referenced segment in • 6–69
specifying private data for • 6–70
Content definition aggregate • 6–69 to 6–70
See also DDIF$_CTD aggregate
external reference index item in • 6–70
external target item in • 6–69
items in • 6–70t, D–12t
label item in • 6–69
private data item in • 6–70
value item in • 6–70
Content reference
specifying label for • 6–22
specifying transformation for • 6–22
Content reference aggregate • 6–22
See also DDIF$_CRF aggregate
items in • 6–22t, D–5t
label item in • 6–22
transformation item in • 6–22
Content stream • 3–16
Conversion
input formats • 2–9 to 2–10
output formats • 2–10 to 2–15
types of • 5–1
CONVERT/DOCUMENT command • 2–3, 2–3 to 2–4
CONVERT AGGREGATE routine • 4–10, CDA–26
CONVERT DOCUMENT routine • 4–10, CDA–29
Converter
activating • CDA–103
calling from an application • CDA–9

Converter (cont'd.)
calling from within an application • 2–3, 2–4 to 2–7
components of • 2–2 to 2–3
format keywords for • 2–3t
Converter kernel • 2–2
CONVERT POSITION routine • 4–11, CDA–31
CONVERT routine • 2–3, 2–4 to 2–7, CDA–9
Copied computed content • 6–38
COPY AGGREGATE routine • 4–5, CDA–33
Counter
specifying style for • 6–79
Counter style aggregate • 6–79 to 6–80
See also DDIF$_CTS aggregate
items in • 6–80t, D–14t
style indicator item in • 6–79
CREATE AGGREGATE routine • 4–4, CDA–35
CREATE FILE routine • 4–2, CDA–37
CREATE ROOT AGGREGATE routine • 4–4, CDA–42
CREATE STREAM routine • 4–3, CDA–46
CREATE TEXT FILE routine • 4–3, CDA–51
Cross-reference computed content • 6–38

D

Data
private • 6–24
Data loss
in DDIF back end • 2–11
in DDIF front end • 2–9
in PostScript back end • 2–12
in Text back end • 2–11
in Text front end • 2–10
Data mapping
in DDIF back end • 2–11
in DDIF front end • 2–9
in PostScript back end • 2–11
in Text back end • 2–11
in Text front end • 2–10
DDDIF$_FTD aggregate
DIF$_FTD_NUMBER item in • 6–70
DDIF$READ_*format* • 5–1, CDA–10, CDA–15
DDIF$WRITE_*format* • 5–1, 5–14, CDA–11, CDA–19
DDIF$_ARC aggregate • 6–18 to 6–20
DDIF$_ARC_CENTER_X item in • 6–18
DDIF$_ARC_CENTER_X_C item in • 6–18
DDIF$_ARC_CENTER_Y item in • 6–18
DDIF$_ARC_CENTER_Y_C item in • 6–18
DDIF$_ARC_EXTENT item in • 6–19
DDIF$_ARC aggregate (cont'd.)
DDIF$_ARC_EXTENT_C item in • 6–19
DDIF$_ARC_FLAGS item in • 6–18
DDIF$_ARC_RADIUS_DELTA_Y item in • 6–19
DDIF$_ARC_RADIUS_X item in • 6–19
DDIF$_ARC_RADIUS_X_C item in • 6–18
DDIF$_ARC_ROTATION item in • 6–19
DDIF$_ARC_ROTATION_C item in • 6–19
DDIF$_ARC_START item in • 6–19
DDIF$_ARC_START_C item in • 6–19
DDIF$_RADIUS_DELTA_Y_C item in • 6–19
items in • 6–19t, D–4t
DDIF$_BEZ aggregate • 6–15 to 6–16
DDIF$_BEZ_FLAGS item in • 6–15
DDIF$_BEZ_PATH item in • 6–16
DDIF$_BEZ_PATH_C item in • 6–16
items in • 6–16t, D–3t
DDIF$_CRF aggregate • 6–22
DDIF$_CRF_REFERENCE item in • 6–22
DDIF$_CRF_TRANSFORM item in • 6–22
items in • 6–22t
DDIF$_CTD aggregate • 6–69 to 6–70
DDIF$_CTD_EXTERNAL_ERF_INDEX item in • 6–70
DDIF$_CTD_EXTERNAL_TARGET item in • 6–69
DDIF$_CTD_LABEL item in • 6–69
DDIF$_CTD_PRIVATE_DATA item in • 6–70
DDIF$_CTD_VALUE item in • 6–70
items in • 6–70t, D–12t
DDIF$_CTS aggregate • 6–79 to 6–80
DDIF$_CTS_STYLE item in • 6–80
DDIF$_CTS_STYLE_C item in • 6–79
items in • 6–80t, D–14t
DDIF$_DDF aggregate • 6–2
DDIF$_DDF_CONTENT item in • 6–2
DDIF$_DDF_DESCRIPTOR item in • 6–2
DDIF$_DDF_HEADER item in • 6–2
DDIF$_DDIF$_SGB aggregate
DDIF$_SGB_CTR_TRIGGER item in • 6–76
DDIF$_DHD aggregate • 6–4 to 6–6
DDIF$_DHD_AUTHOR item in • 6–4
DDIF$_DHD_CONFORMANCE_TAGS item in • 6–4
DDIF$_DHD_DATE item in • 6–4
DDIF$_DHD_EXTERNAL_REFERENCES item in • 6–4
DDIF$_DHD_LANGUAGES item in • 6–5
DDIF$_DHD_LANGUAGES_C item in • 6–5
DDIF$_DHD_PRIVATE_DATA item in • 6–4
DDIF$_DHD_STYLE_GUIDE item in • 6–5
DDIF$_DHD_TITLE item in • 6–4
DDIF$_DHD_VERSION item in • 6–4

DDIF$_DSC aggregate • 6–3 to 6–4
DDIF$_DSC_MAJOR_VERSION item in • 6–3
DDIF$_DSC_MINOR_VERSION item in • 6–3
DDIF$_DSC_PRODUCT_IDENTIFIER item in • 6–3
DDIF$_DSC_PRODUCT_NAME item in • 6–3
DDIF$_ERF aggregate • 6–28 to 6–29
DDIF$_ERF_CONTROL item in • 6–29
DDIF$_ERF_DATA_TYPE item in • 6–28
DDIF$_ERF_DESCRIPTOR item in • 6–28
DDIF$_ERF_LABEL item in • 6–29
DDIF$_ERF_LABEL_TYPE item in • 6–29
items in • 6–29t, D–7t
DDIF$_EXT aggregate • 6–22 to 6–23
DDIF$_EXT_DATA_VALUE_DESCRIPTOR item in • 6–23
DDIF$_EXT_DIRECT_REFERENCE item in • 6–22
DDIF$_EXT_ENCODING item in • 6–23
DDIF$_EXT_ENCODING_C item in • 6–23
DDIF$_EXT_ENCODING_L item in • 6–23
DDIF$_EXT_INDIRECT_REFERENCE item in • 6–22
items in • 6–23t, D–5t
DDIF$_FAS aggregate • 6–20 to 6–21
DDIF$_FAS_FLAGS item in • 6–20
DDIF$_FAS_PATH item in • 6–20
items in • 6–21t, D–4t
DDIF$_FTD aggregate • 6–70 to 6–71
DDIF$_FTD_IDENTIFIER item in • 6–70
DDIF$_FTD_PRIVATE_DATA item in • 6–70
items in • 6–71t, D–12t
DDIF$_GLA aggregate • 6–91 to 6–93
DDIF$_GLA_BOTTOM_MARGIN item in • 6–92
DDIF$_GLA_BOTTOM_MARGIN_C item in • 6–92
DDIF$_GLA_LEFT_MARGIN item in • 6–92
DDIF$_GLA_LEFT_MARGIN_C item in • 6–92
DDIF$_GLA_RIGHT_MARGIN item in • 6–92
DDIF$_GLA_RIGHT_MARGIN_C item in • 6–92
DDIF$_GLA_TOP_MARGIN item in • 6–91
DDIF$_GLA_TOP_MARGIN_C item in • 6–91
items in • 6–92t, D–16t
DDIF$_GLY aggregate • 6–25 to 6–28
DDIF$_GLY_BOUNDING_BOX_LL_X item in • 6–26
DDIF$_GLY_BOUNDING_BOX_LL_X_C item in • 6–26
DDIF$_GLY_BOUNDING_BOX_LL_Y item in • 6–26
DDIF$_GLY_BOUNDING_BOX_LL_Y_C item in • 6–26
DDIF$_GLY aggregate (cont'd.)
DDIF$_GLY_BOUNDING_BOX_UR_X item in • 6–26
DDIF$_GLY_BOUNDING_BOX_UR_X_C item in • 6–26
DDIF$_GLY_BOUNDING_BOX_UR_Y item in • 6–26
DDIF$_GLY_BOUNDING_BOX_UR_Y_C item in • 6–26
DDIF$_GLY_FLAGS item in • 6–26
DDIF$_GLY_ID item in • 6–26
DDIF$_GLY_OUTLINE item in • 6–26
DDIF$_GLY_STREAMS item in • 6–27
DDIF$_GLY_SUCCESSOR item in • 6–27
DDIF$_GLY_SUCCESSOR_C item in • 6–27
items in • 6–27t, D–6t
DDIF$_GTX aggregate • 6–10
DDIF$_GTX_CONTENT item in • 6–10
DDIF$_HRD aggregate • 6–11
DDIF$_HRD_DIRECTIVE item in • 6–11
DDIF$_HRV aggregate • 6–13 to 6–14
DDIF$_HRV_C item in • 6–13
DDIF$_HRV_ESC_CONSTANT item in • 6–13
DDIF$_HRV_ESC_CONSTANT_C item in • 6–13
DDIF$_HRV_ESC_RATIO_D item in • 6–13
DDIF$_HRV_ESC_RATIO_N item in • 6–13
DDIF$_HRV_RESET_VALUE item in • 6–13
DDIF$_HRV_RESET_VALUE_C item in • 6–13
DDIF$_HRV_RESET_VARIABLE item in • 6–13
DDIF$_IDU aggregate • 6–30 to 6–31
DDIF$_IDU_BITS_PER_PIXEL item in • 6–31
DDIF$_IDU_COMPRESSION_PARAMS item in • 6–30
DDIF$_IDU_COMPRESSION_TYPE item in • 6–30
DDIF$_IDU_DATA_OFFSET item in • 6–30
DDIF$_IDU_NUMBER_OF_LINES item in • 6–30
DDIF$_IDU_PIXELS_PER_LINE item in • 6–30
DDIF$_IDU_PIXEL_ORDER item in • 6–31
DDIF$_IDU_PIXEL_STRIDE item in • 6–30
DDIF$_IDU_PLANE_DATA item in • 6–31
DDIF$_IDU_PRIVATE_CODING_ATTR item in • 6–30
DDIF$_IDU_SCANLINE_STRIDE item in • 6–30
items in • 6–31t, D–7t
DDIF$_IMG aggregate • 6–21
DDIF$_IMG_CONTENT item in • 6–21
items in • 6–21t, D–5t
DDIF$_LG1 aggregate • 6–84 to 6–85
DDIF$_LG1_PAGE_DESCRIPTIONS item in • 6–84
DDIF$_LG1_PRIVATE_DATA item in • 6–84

DDIF$_LG1 aggregate (cont'd.)
items in • 6–84t, D–15t
DDIF$_LIN aggregate • 6–16 to 6–18
DDIF$_LIN_DRAW_PATTERN item in • 6–17
DDIF$_LIN_FLAGS item in • 6–16
DDIF$_LIN_PATH item in • 6–17
DDIF$_LIN_PATH_C item in • 6–17
items in • 6–18t, D–4t
DDIF$_LL1 aggregate • 6–88 to 6–91
DDIF$_LL1_BREAK_AFTER item in • 6–89
DDIF$_LL1_BREAK_BEFORE item in • 6–89
DDIF$_LL1_BREAK_WITHIN item in • 6–89
DDIF$_LL1_GALLEY_SELECT item in • 6–89
DDIF$_LL1_INITIAL_DIRECTIVE item in • 6–88
DDIF$_LL1_INITIAL_INDENT item in • 6–89
DDIF$_LL1_INITIAL_INDENT_C item in • 6–89
DDIF$_LL1_LEADING_CONSTANT item in • 6–90
DDIF$_LL1_LEADING_CONSTANT_C item in • 6–90
DDIF$_LL1_LEADING_RATIO_D item in • 6–90
DDIF$_LL1_LEADING_RATIO_N item in • 6–90
DDIF$_LL1_LEFT_INDENT item in • 6–89
DDIF$_LL1_LEFT_INDENT_C item in • 6–89
DDIF$_LL1_RIGHT_INDENT item in • 6–90
DDIF$_LL1_RIGHT_INDENT_C item in • 6–90
DDIF$_LL1_SPACE_AFTER item in • 6–90
DDIF$_LL1_SPACE_AFTER_C item in • 6–90
DDIF$_LL1_SPACE_BEFORE item in • 6–90
DDIF$_LL1_SPACE_BEFORE_C item in • 6–90
DDIF$_LL1_TAB_STOPS item in • 6–91
items in • 6–91t, D–16t
DDIF$_LS1 aggregate • 6–85 to 6–86
DDIF$_LS1_LAYOUT item in • 6–85
DDIF$_LS1_LAYOUT_C item in • 6–85
items in • 6–85t, D–15t
DDIF$_LSD aggregate • 6–71 to 6–72
DDIF$_LSD_NUMBER item in • 6–71
DDIF$_LSD_PATTERN item in • 6–53, 6–71
DDIF$_LSD_PRIVATE_DATA item in • 6–72
items in • 6–72t, D–12t
DDIF$_LW1 aggregate • 6–86 to 6–88
DDIF$_LW1_HYPHENATION_FLAGS item in • 6–87
DDIF$_LW1_HYPH_LINES item in • 6–87
DDIF$_LW1_MAXIMUM_ORPHAN_SIZE item in • 6–87
DDIF$_LW1_MAXIMUM_WIDOW_SIZE item in • 6–87
DDIF$_LW1_QUAD_FORMAT item in • 6–86
DDIF$_LW1_WRAP_FORMAT item in • 6–86
items in • 6–88t, D–15t
DDIF$_OCC aggregate • 6–80 to 6–81
DDIF$_OCC aggregate (cont'd.)
DDIF$_OCC_OCCURRENCE_C item in • 6–80
DDIF$_OCC_STRUCTURE_ELEMENT item in • 6–81
DDIF$_OCC_STRUCTURE_ELEMENT_C item in • 6–80
items in • 6–81t, D–14t
DDIF$_PGD aggregate • 6–93 to 6–94
DDIF$_PGD_DESC item in • 6–93
DDIF$_PGD_DESC_C item in • 6–93
DDIF$_PGD_LABEL item in • 6–93
DDIF$_PGD_PRIVATE_DATA item in • 6–93
items in • 6–93t, D–17t
DDIF$_PGL aggregate • 6–94 to 6–96
DDIF$_PGL_CONTENT item in • 6–95
DDIF$_PGL_LAYOUT_ID item in • 6–94
DDIF$_PGL_ORIENTATION item in • 6–95
DDIF$_PGL_PROTOTYPE item in • 6–95
DDIF$_PGL_SIZE_X_NOM item in • 6–94
DDIF$_PGL_SIZE_X_NOM_C item in • 6–94
DDIF$_PGL_SIZE_X_SHR item in • 6–94
DDIF$_PGL_SIZE_X_SHR_C item in • 6–94
DDIF$_PGL_SIZE_X_STR item in • 6–94
DDIF$_PGL_SIZE_X_STR_C item in • 6–94
DDIF$_PGL_SIZE_Y_NOM item in • 6–94
DDIF$_PGL_SIZE_Y_NOM_C item in • 6–94
DDIF$_PGL_SIZE_Y_SHR item in • 6–95
DDIF$_PGL_SIZE_Y_SHR_C item in • 6–95
DDIF$_PGL_SIZE_Y_STR item in • 6–95
DDIF$_PGL_SIZE_Y_STR_C item in • 6–94
items in • 6–95t, D–17t
DDIF$_PGS aggregate • 6–96 to 6–97
DDIF$_PGS_SELECT_PAGE_LAYOUT item in • 6–97
DDIF$_PGS_SELECT_PAGE_LAYOUT_C item in • 6–96
DDIF$_PGS_SIDE_CRITERIA item in • 6–96
items in • 6–97t, D–18t
DDIF$_PHD aggregate • 6–72 to 6–73
DDIF$_PHD_DESCRIPTION item in • 6–72
DDIF$_PHD_NUMBER item in • 6–72
DDIF$_PHD_PRIVATE_DATA item in • 6–72
items in • 6–72t, D–13t
DDIF$_PTD aggregate • 6–73 to 6–75
DDIF$_PTD_DEFN_C item in • 6–73
DDIF$_PTD_NUMBER item in • 6–73
DDIF$_PTD_PAT_COLORS item in • 6–74
DDIF$_PTD_PAT_NUMBER item in • 6–74
DDIF$_PTD_PRIVATE_DATA item in • 6–74
DDIF$_PTD_RAS_PATTERN item in • 6–74
DDIF$_PTD_SOL_COLOR_B item in • 6–74
DDIF$_PTD_SOL_COLOR_C item in • 6–73

DDIF$_PTD aggregate (cont'd.)
DDIF$_PTD_SOL_COLOR_G item in • 6–74
DDIF$_PTD_SOL_COLOR_R item in • 6–73
items in • 6–74t, D–13t
DDIF$_PTH aggregate • 6–32 to 6–34
DDIF$_PTH_ARC_CENTER_X item in • 6–33
DDIF$_PTH_ARC_CENTER_X_C item in • 6–33
DDIF$_PTH_ARC_CENTER_Y item in • 6–33
DDIF$_PTH_ARC_CENTER_Y_C item in • 6–33
DDIF$_PTH_ARC_EXTENT item in • 6–33
DDIF$_PTH_ARC_EXTENT_C item in • 6–33
DDIF$_PTH_ARC_RADIUS_DELTA_Y item in • 6–33
DDIF$_PTH_ARC_RADIUS_DELTA_Y_C item in • 6–33
DDIF$_PTH_ARC_RADIUS_X item in • 6–33
DDIF$_PTH_ARC_RADIUS_X_C item in • 6–33
DDIF$_PTH_ARC_ROTATION item in • 6–34
DDIF$_PTH_ARC_ROTATION_C item in • 6–34
DDIF$_PTH_ARC_START item in • 6–33
DDIF$_PTH_ARC_START_C item in • 6–33
DDIF$_PTH_BEZ_PATH item in • 6–32
DDIF$_PTH_BEZ_PATH_C item in • 6–32
DDIF$_PTH_C item in • 6–32
DDIF$_PTH_LIN_PATH item in • 6–32
DDIF$_PTH_LIN_PATH_C item in • 6–32
DDIF$_PTH_REFERENCE item in • 6–34
items in • 6–34t, D–7t
DDIF$_PVT aggregate • 6–24 to 6–25
DDIF$_PVT_DATA item in • 6–25
DDIF$_PVT_DATA_C item in • 6–24
DDIF$_PVT_NAME item in • 6–24
DDIF$_PVT_REFERENCE_ERF_INDEX item in • 6–25
items in • 6–25t, D–6t
DDIF$_RCD aggregate • 6–81 to 6–82
DDIF$_RCD_CONTENTS item in • 6–81
DDIF$_RCD_TAG item in • 6–81
DDIF$_RCD_TYPE item in • 6–81
items in • 6–82t, D–14t
DDIF$_RGB aggregate • 6–82
DDIF$_RGB_BLUE_VALUE item in • 6–82
DDIF$_RGB_GREEN_VALUE item in • 6–82
DDIF$_RGB_LUT_INDEX item in • 6–82
DDIF$_RGB_RED_VALUE item in • 6–82
items in • 6–82t, D–15t
DDIF$_SEG aggregate • 6–6 to 6–8
DDIF$_SEG_CONTENT item in • 6–7
DDIF$_SEG_GENERIC_LAYOUT item in • 6–7
DDIF$_SEG_ID item in • 6–6
DDIF$_SEG_SEGMENT_TYPE item in • 6–6
DDIF$_SEG aggregate (cont'd.)
DDIF$_SEG_SPECIFIC_ATTRIBUTES item in • 6–7
DDIF$_SEG_SPECIFIC_LAYOUT item in • 6–7
DDIF$_SEG_USER_LABEL item in • 6–6
DDIF$_SFT aggregate • 6–11
DDIF$_SFT_DIRECTIVE item in • 6–11
DDIF$_SFV aggregate • 6–14 to 6–15
DDIF$_SFV_C item in • 6–14
DDIF$_SFV_ESC_CONSTANT item in • 6–14
DDIF$_SFV_ESC_CONSTANT_C item in • 6–14
DDIF$_SFV_ESC_RATIO_D item in • 6–14
DDIF$_SFV_ESC_RATIO_N item in • 6–14
DDIF$_SFV_RESET_VALUE item in • 6–15
DDIF$_SFV_RESET_VALUE_C item in • 6–15
DDIF$_SFV_RESET_VARIABLE item in • 6–15
DDIF$_SGA aggregate • 6–35 to 6–69
DDIF$_SGA_ALT_PRESENTATION item in • 6–41
DDIF$_SGA_BINDING_DEFNS item in • 6–36
DDIF$_SGA_COMPUTE_C item in • 6–37
DDIF$_SGA_CONTENT_CATEGORY item in • 6–8, 6–36
DDIF$_SGA_CONTENT_DEFNS item in • 6–47
DDIF$_SGA_CONTENT_STREAMS item in • 6–36
DDIF$_SGA_CPTCPY_ERF_INDEX item in • 6–38
DDIF$_SGA_CPTCPY_TARGET item in • 6–38
DDIF$_SGA_CPTFNC_NAME item in • 6–38
DDIF$_SGA_CPTFNC_PARAMETERS item in • 6–39
DDIF$_SGA_CPTVAR_VARIABLE item in • 6–38
DDIF$_SGA_CPTXRF_ERF_INDEX item in • 6–38
DDIF$_SGA_CPTXRF_TARGET item in • 6–38
DDIF$_SGA_CPTXRF_VARIABLE item in • 6–38
DDIF$_SGA_FONT_DEFNS item in • 6–47
DDIF$_SGA_FRMFXD_POSITION_X item in • 6–63
DDIF$_SGA_FRMFXD_POSITION_X_C item in • 6–63
DDIF$_SGA_FRMFXD_POSITION_Y item in • 6–63
DDIF$_SGA_FRMFXD_POSITION_Y_C item in • 6–63
DDIF$_SGA_FRMGLY_HORIZONTAL item in • 6–64
DDIF$_SGA_FRMGLY_VERTICAL item in • 6–63
DDIF$_SGA_FRMINL_BASE_OFFSET item in • 6–63
DDIF$_SGA_FRMINL_BASE_OFFSET_C item in • 6–63

DDIF$_SGA aggregate (cont'd.)
DDIF$_SGA_FRMMAR_BASE_OFFSET item in • 6–64
DDIF$_SGA_FRMMAR_BASE_OFFSET_C item in • 6–64
DDIF$_SGA_FRMMAR_HORIZONTAL item in • 6–64
DDIF$_SGA_FRMMAR_NEAR_OFFSET item in • 6–64
DDIF$_SGA_FRMMAR_NEAR_OFFSET_C item in • 6–64
DDIF$_SGA_FRM_BOX_LL_X item in • 6–61
DDIF$_SGA_FRM_BOX_LL_X_C item in • 6–61
DDIF$_SGA_FRM_BOX_LL_Y item in • 6–61
DDIF$_SGA_FRM_BOX_LL_Y_C item in • 6–61
DDIF$_SGA_FRM_BOX_UR_X item in • 6–61
DDIF$_SGA_FRM_BOX_UR_X_C item in • 6–61
DDIF$_SGA_FRM_BOX_UR_Y item in • 6–61
DDIF$_SGA_FRM_BOX_UR_Y_C item in • 6–61
DDIF$_SGA_FRM_CLIPPING item in • 6–62
DDIF$_SGA_FRM_FLAGS item in • 6–60
DDIF$_SGA_FRM_OUTLINE item in • 6–61
DDIF$_SGA_FRM_POSITION_C item in • 6–62
DDIF$_SGA_FRM_TRANSFORM item in • 6–65
DDIF$_SGA_GLY_ATTRIBUTES item in • 6–55
DDIF$_SGA_IMG_BITS_PER_COMP item in • 6–60
DDIF$_SGA_IMG_BRT_POLARITY item in • 6–56
DDIF$_SGA_IMG_COMP_SPACE_ORG item in • 6–58
DDIF$_SGA_IMG_COMP_WAVELENGTH item in • 6–58
DDIF$_SGA_IMG_COMP_WAVELENGTH_C item in • 6–58
DDIF$_SGA_IMG_GRID_TYPE item in • 6–56
DDIF$_SGA_IMG_LINE_PROGRESSION item in • 6–56
DDIF$_SGA_IMG_LOOKUP_TABLES item in • 6–57
DDIF$_SGA_IMG_LOOKUP_TABLES_C item in • 6–57
DDIF$_SGA_IMG_LP_PIXEL_DIST item in • 6–56
DDIF$_SGA_IMG_NUMBER_OF_COMP item in • 6–60
DDIF$_SGA_IMG_PIXEL_PATH item in • 6–56
DDIF$_SGA_IMG_PLANES_PER_PIXEL item in • 6–59
DDIF$_SGA_IMG_PLANE_SIGNIF item in • 6–59
DDIF$_SGA_IMG_PP_PIXEL_DIST item in • 6–56
DDIF$_SGA_IMG_PRIVATE_DATA item in • 6–56
DDIF$_SGA_IMG_SPECTRAL_MAPPING item in • 6–57
DDIF$_SGA aggregate (cont'd.)
DDIF$_SGA_IMG_TIMING_DESC item in • 6–57
DDIF$_SGA_ITEM_CHANGE_LIST item in • 6–65
DDIF$_SGA_LANGUAGE item in • 6–39
DDIF$_SGA_LAYGLY_LAYOUT item in • 6–42
DDIF$_SGA_LAYGLY_WRAP item in • 6–42
DDIF$_SGA_LAYOUT_C item in • 6–41
DDIF$_SGA_LAYPOS_TEXT_POSITION item in • 6–46
DDIF$_SGA_LAYPTH_FORMAT item in • 6–42
DDIF$_SGA_LAYPTH_H_ALIGN item in • 6–44
DDIF$_SGA_LAYPTH_ORIENTATION item in • 6–44
DDIF$_SGA_LAYPTH_ORIENTATION_C item in • 6–43
DDIF$_SGA_LAYPTH_PATH item in • 6–42
DDIF$_SGA_LAYPTH_V_ALIGN item in • 6–44
DDIF$_SGA_LAYREL_H_CONSTANT item in • 6–45
DDIF$_SGA_LAYREL_H_CONSTANT_C item in • 6–45
DDIF$_SGA_LAYREL_H_RATIO_D item in • 6–45
DDIF$_SGA_LAYREL_H_RATIO_N item in • 6–45
DDIF$_SGA_LAYREL_V_CONSTANT item in • 6–46
DDIF$_SGA_LAYREL_V_CONSTANT_C item in • 6–46
DDIF$_SGA_LAYREL_V_RATIO_D item in • 6–46
DDIF$_SGA_LAYREL_V_RATIO_N item in • 6–45
DDIF$_SGA_LEGEND_UNIT_D item in • 6–40
DDIF$_SGA_LEGEND_UNIT_NAME item in • 6–40
DDIF$_SGA_LEGEND_UNIT_N item in • 6–40
DDIF$_SGA_LINE_STYLE_DEFNS item in • 6–47
DDIF$_SGA_LIN_END_FINISH item in • 6–53
DDIF$_SGA_LIN_END_SIZE item in • 6–54
DDIF$_SGA_LIN_END_SIZE_C item in • 6–54
DDIF$_SGA_LIN_END_START item in • 6–53
DDIF$_SGA_LIN_INTERIOR_PATTERN item in • 6–54
DDIF$_SGA_LIN_JOIN item in • 6–54
DDIF$_SGA_LIN_MASK_PATTERN item in • 6–53
DDIF$_SGA_LIN_MITER_LIMIT_D item in • 6–54
DDIF$_SGA_LIN_MITER_LIMIT_N item in • 6–54
DDIF$_SGA_LIN_PATTERN_SIZE item in • 6–52
DDIF$_SGA_LIN_PATTERN_SIZE_C item in • 6–52
DDIF$_SGA_LIN_STYLE item in • 6–52
DDIF$_SGA_LIN_WIDTH item in • 6–52
DDIF$_SGA_LIN_WIDTH_C item in • 6–52
DDIF$_SGA_MKR_MASK_PATTERN item in • 6–55

DDIF$_SGA aggregate (cont'd.)
DDIF$_SGA_MKR_SIZE item in • 6–55
DDIF$_SGA_MKR_SIZE_C item in • 6–55
DDIF$_SGA_MKR_STYLE item in • 6–55
DDIF$_SGA_PATH_DEFNS item in • 6–47
DDIF$_SGA_PATTERN_DEFNS item in • 6–47
DDIF$_SGA_PRIVATE_DATA item in • 6–35
DDIF$_SGA_SEGMENT_TAGS item in • 6–36
DDIF$_SGA_STRUCTURE_DESC item in • 6–39
DDIF$_SGA_STRUCTURE_DESC_C item in • 6–39
DDIF$_SGA_TXT_DEC_ALIGNMENT item in • 6–51
DDIF$_SGA_TXT_DIRECTION item in • 6–51
DDIF$_SGA_TXT_FONT item in • 6–48
DDIF$_SGA_TXT_HEIGHT item in • 6–50
DDIF$_SGA_TXT_HEIGHT_C item in • 6–50
DDIF$_SGA_TXT_LEADER_ALIGN item in • 6–51
DDIF$_SGA_TXT_LEADER_BULLET item in • 6–51
DDIF$_SGA_TXT_LEADER_SPACE item in • 6–51
DDIF$_SGA_TXT_LEADER_SPACE_C item in • 6–51
DDIF$_SGA_TXT_LEADER_STYLE item in • 6–51
DDIF$_SGA_TXT_MASK_PATTERN item in • 6–48
DDIF$_SGA_TXT_PAIR_KERNING item in • 6–52
DDIF$_SGA_TXT_RENDITION item in • 6–49
DDIF$_SGA_TXT_SET_SIZE_D item in • 6–50
DDIF$_SGA_TXT_SET_SIZE_N item in • 6–50
DDIF$_SGA_TYPE_DEFNS item in • 6–48
DDIF$_SGA_UNITS_PER_MEASURE item in • 6–40
DDIF$_SGA_UNIT_NAME item in • 6–40
items in • 6–66t, D–8t
DDIF$_SGB aggregate • 6–75 to 6–78
DDIF$_SGB_COM_STRING_EXPR item in • 6–77
DDIF$_SGB_COM_STRING_EXPR_C item in • 6–77
DDIF$_SGB_CTR_INIT item in • 6–76
DDIF$_SGB_CTR_INIT_C item in • 6–76
DDIF$_SGB_CTR_STYLE item in • 6–76
DDIF$_SGB_CTR_TRIGGER_C item in • 6–76
DDIF$_SGB_CTR_TYPE item in • 6–77
DDIF$_SGB_RCD_LIST item in • 6–77
DDIF$_SGB_VARIABLE_NAME item in • 6–75
DDIF$_SGB_VARIABLE_VALUE_C item in • 6–75
items in • 6–78t, D–13t
DDIF$_TBS aggregate • 6–97 to 6–98
DDIF$_TBS aggregate (cont'd.)
DDIF$_TBS_HORIZONTAL_POSITION item in • 6–97
DDIF$_TBS_HORIZONTAL_POSITION_C item in • 6–97
DDIF$_TBS_LEADER item in • 6–98
DDIF$_TBS_TYPE item in • 6–97
items in • 6–98t, D–18t
DDIF$_TRN aggregate • 6–83 to 6–84
DDIF$_TRN_PARAMETER item in • 6–84
DDIF$_TRN_PARAMETER_C item in • 6–83
items in • 6–84t, D–15t
DDIF$_TXT aggregate • 6–10
DDIF$_TXT_CONTENT item in • 6–10
DDIF$_TYD aggregate • 6–78 to 6–79
DDIF$_TYD_ATTRIBUTES item in • 6–78
DDIF$_TYD_LABEL item in • 6–78
DDIF$_TYD_PARENT item in • 6–78
DDIF$_TYD_PRIVATE_DATA item in • 6–78
items in • 6–79t, D–14t
DDIF (DIGITAL Document Interchange Format)
analyzing files encoded in • 2–15
VMS RMS support of • A–1
DDIF back end • 2–10 to 2–11
data loss in • 2–11
data mapping in • 2–11
DDIF front end • 2–9
data loss in • 2–9
data mapping in • 2–9
document syntax errors in • 2–9
external file references in • 2–9
DDIF-to-Text RMS extension • A–1
DDIF viewer • 2–7
DDIF_CRF aggregate
items in • D–5t
DDIS encoding
definition of • 4–8
Deallocation routine • CDA–40, CDA–44, CDA–48, CDA–110, CDA–113
DELETE AGGREGATE routine • 4–5, CDA–54
DELETE ROOT AGGREGATE routine • 4–4, CDA–56
DIGITAL Document Interchange Format
See DDIF
Directive • 6–11 to 6–15
hard • 6–11
hard value • 6–13 to 6–14
soft • 6–11
soft value • 6–14 to 6–15
values for • 6–12t
Document
See also Compound document

Document (cont'd.)
converting • 4–10, 5–1 to 5–2
creating for output • CDA–37
definition of • 4–1
describing the encoding of • 6–2
distinguishing versions of • 6–4
final form • 1–2
hierarchy of • 3–2 to 3–8
indicating the name of • 6–3
reading • 4–10, CDA–29
reading from a stream • CDA–82
representing the encoding software of • 6–3
returning position in • CDA–121
returning size of • CDA–121
revisable • 1–1
relationships in • 3–6 to 3–7
specifying external style guide for • 6–5
specifying file references in • 6–4
specifying parameters for • 6–2
specifying private information for • 6–4
specifying processing languages in • 6–5
specifying processing restrictions for • 6–4
specifying the author of • 6–4
specifying the content of • 6–2
specifying the title of • 6–4
specifying version date of • 6–4
structured • 1–4
structure of • 6–1, 6–1f
testing the compatibility of versions for • 6–3
types of • 1–1 to 1–2
writing • 4–10, CDA–126
Document content
See Content
Document content aggregate • 6–6 to 6–8
See also DDIF$_SEG aggregate
generic layout item in • 6–7
items in • 6–7t, D–2t
segment attribute item in • 6–7
segment content item in • 6–7
segment identifier item in • 6–6
segment type item in • 6–6
segment user label item in • 6–6
specific layout item in • 6–7
Document conversion
types of • 5–1
Document descriptor • 3–4
Document descriptor aggregate • 6–3 to 6–4
See also DDIF$_DSC aggregate
items in • 6–3t, D–1t
major version item in • 6–3
minor version item in • 6–3
Document descriptor aggregate (cont'd.)
product identifier item in • 6–3
product name item in • 6–3
Document format
see Layout
Document header • 3–4
Document header aggregate • 6–4 to 6–6
See also DDIF$_DHD aggregate
author item in • 6–4
conformance tags item in • 6–4
date item in • 6–4
external references item in • 6–4
items in • 6–5t, D–1t
language item in • 6–5
languages indicator item in • 6–5
private header data item in • 6–4
style guide item in • 6–5
title item in • 6–4
version item in • 6–4
Document hierarchy • 3–2 to 3–8
Document layout
See Layout
Document-method conversion • 5–8 to 5–10
Close • 5–10
DDIF$READ_*format* • 5–9 to 5–10
Get-Aggregate • 5–10
Get-Position • 5–10
Document root • 3–3
Document scope • 4–12 to 4–13
completing • CDA–94
entering • CDA–57
Document segment
See Segment
Document syntax errors
in DDIF front end • 2–9
in Text front end • 2–10
Document transfer • 4–9 to 4–10
determining position in • 4–10, 4–11
using an entire document • 4–10

E

ENTER SCOPE routine • 4–12, CDA–57
Enumeration
AngleRef • 4–7
encoding of • 4–6
expression • 4–8
measurement • 4–7
ERASE ITEM routine • 4–9, CDA–68

Escapement type • E–19
Expression enumeration • 4–8
External reference
  identifying data type of • 6–28
  identifying storage system of • 6–29
  in DDIF front end • 2–9
  in Text front end • 2–10
  processing • 4–13, CDA–103
  specifying description of the data type of • 6–28
  specifying label for • 6–29
  specifying treatment of • 6–29
External reference aggregate • 6–28 to 6–29
  See also DDIF$_ERF aggregate
  control item in • 6–29
  items in • 6–29t, D–7t
  reference data type item in • 6–28
  reference descriptor item in • 6–28
  reference label item in • 6–29
  storage item in • 6–29
External restricted content aggregate • 6–22 to 6–23
  See also DDIF$_EXT aggregate
  data value descriptor item in • 6–23
  direct reference item in • 6–22
  encoding indicator item in • 6–23
  encoding length item in • 6–23
  indirect reference item in • 6–22
  items in • 6–23t, D–5t

F

File
  See also Text file
  closing • 4–2, CDA–5
  creating • 4–2, CDA–37
  opening • 4–2, CDA–106
Files
  specifying processing options during conversion of • 2–8
File tag
  accessing • A–10
  creation of • A–1
  DDIF • A–1
  disposition by COPY command • A–4
  preserving • A–13
  requirement for • A–1
  use of • A–1
Fill area set
  controlling the rendition of • 6–20
Fill area set (cont'd.)
  specifying the composite path of • 6–20
Fill area set content aggregate • 6–20 to 6–21
  See also DDIF$_FAS aggregate
  flags item in • 6–20
  items in • 6–21t, D–4t
  set path item in • 6–20
Final form document • 1–2
FIND DEFINITION routine • CDA–70
FIND TRANSFORMATION routine • CDA–73
Flush routine • CDA–76
FLUSH STREAM routine • 4–3, CDA–75
Font definition
  specifying for the defining segment • 6–70
  specifying name for • 6–70
  specifying private data for • 6–70
Font definition aggregate • 6–70 to 6–71
  See also DDIF$_FTD aggregate
  identifier item in • 6–70
  items in • 6–71t, D–12t
  number item in • 6–70
  private data item in • 6–70
Frame
  controlling presentation of • 6–60
  fixed position • 6–63
  galley • 6–63
  inline position • 6–63
  margin • 6–64
  origin of • 3–5
  specifying a coordinate transformation for • 6–65
  specifying attributes of • 6–60 to 6–65
  specifying lower left corner *x* position of • 6–61
  specifying lower left corner *y* position of • 6–61
  specifying the clipping path of • 6–62
  specifying the horizontal offset of the base for • 6–64
  specifying the horizontal positioning of • 6–64
  specifying the horizontal position of the lower left corner of • 6–64
  specifying the outline path of • 6–61
  specifying the position of • 6–62
  specifying the vertical offset from the base for • 6–64
  specifying the vertical offset of the origin of • 6–63
  specifying the vertical positioning of the lower edge of • 6–63
  specifying the *x* position of the origin of • 6–63
  specifying the *y* position of the origin of • 6–63
  specifying upper right corner *x* position of • 6–61
  specifying upper right corner *y* position of • 6–61
Frame-based layout • 6–41

Front end • 5–2 to 5–13, CDA–15
aggregate-method conversion • 5–10 to 5–13
Close entry point • 5–8
DDIF • 2–9
DDIF$READ_*format* entry point • 5–4 to 5–6
document-method conversion • 5–8 to 5–10
entry point • CDA–10
Get-Aggregate entry point • 5–6 to 5–7
Get-Position entry point • 5–7 to 5–8
invoking during conversion • 4–13
text • 2–10
Function computed content • 6–38

G

Galley • 3–15
specifying attributes for • 6–55
specifying bottom margin for • 6–92
Specifying content streams for • 3–16
specifying left margin for • 6–92
specifying right margin for • 6–92
specifying top margin for • 6–91
Galley attributes aggregate • 6–91 to 6–93
See also DDIF$_GLA aggregate
galley bottom margin item in • 6–92
galley left margin item in • 6–92
galley right margin item in • 6–92
galley top margin item in • 6–91
items in • 6–92t, D–16t
Galley-based layout • 6–41, 6–42
General text content • 6–10
General text content aggregate • 6–10
See also DDIF$_GTX aggregate
text content item in • 6–10
Generic content • 3–8 to 3–9
referencing • 3–9
Generic layout
specifying descriptions of page templates and rules for • 6–84
specifying private data in • 6–84
Generic layout aggregate • 6–84 to 6–85
See also DDIF$_LG1 aggregate
items in • 6–84t, D–15t
page descriptions item in • 6–84
private data item in • 6–84
Generic type • 3–8
referencing • 3–9
GET AGGREGATE routine • 4–11, CDA–77
GET ARRAY SIZE routine • 4–9, CDA–80
GET DOCUMENT routine • 4–10, CDA–82
GET EXTERNAL ENCODING routine • CDA–84
Get routine • 5–13, CDA–114
GET STREAM POSITION routine • CDA–86
GET TEXT POSITION routine • CDA–89
Graphics
controlling interior fill pattern for • 6–54
Graphics content • 3–1, 3–5

H

Handle
definition of • 4–1
Hard content • 3–1
Hard directive • 6–11
values for • 6–12t
Hard directive aggregate • 6–11
See also DDIF$_HRD aggregate
hard directive item in • 6–11
Hard value directive • 6–13 to 6–14
specifying escapement constant for • 6–13
specifying escapement ratio denominator for • 6–13
specifying escapement ratio numerator for • 6–13
specifying new variable value for • 6–13
specifying type of • 6–13
specifying variable to be reset by • 6–13
Hard value directive aggregate • 6–13 to 6–14
See also DDIF$_HRV aggregate
directive choice item in • 6–13
escapement constant indicator in • 6–13
escapement ratio item in • 6–13
items in • 6–14t, D–3t
reset value item in • 6–13
reset variable item in • 6–13

I

Image
specifying application-private lookup tables for • 6–57
specifying aspect ratio along the pixel path of • 6–56
specifying attributes for • 6–56 to 6–58
specifying correlation between physical image data and spectral components of • 6–57
specifying direction of scanline capture for • 6–56

Image (cont'd.)
specifying line progression path aspect ratio for • 6–56
specifying motion sequence in • 6–57
specifying private data for • 6–56
specifying the contents of • 6–21
specifying the direction of pixel capture path for • 6–56
specifying the physical format of the pixel grid of • 6–56
specifying the representation of intensity levels in • 6–56
specifying wavelength information for • 6–58
Image component space
specifying attributes for • 6–58 to 6–60
specifying number of bits used for each image in • 6–60
specifying number of data planes for pixel in • 6–59
specifying number of spectral components in • 6–60
specifying physical organization of • 6–58
specifying significance of data planes in • 6–59
Image content • 3–1, 3–5, 6–21 to 6–22
Image content aggregate • 6–21
image content item in • 6–21
items in • 6–21t, D–5t
See also DDIF$_IMG aggregate • 6–21
Image data • 6–21
containing parameters for compression of • 6–30
indicating compression scheme for a plane of • 6–30
specifying actual values of • 6–31
specifying distance between pixels in • 6–30
specifying distance between scanlines in • 6–30
specifying number of pixels per scanline in • 6–30
specifying number of scanlines in • 6–30
specifying offset to first bit of • 6–30
specifying pixel order in • 6–31
specifying private coding attributes for • 6–30
specifying total number of bits per pixel in • 6–31
Image data unit aggregate • 6–30 to 6–31
See also DDIF$_IDU aggregate
compression parameters item in • 6–30
compression type item in • 6–30
data offset item in • 6–30
items in • 6–31t, D–7t
number of lines item in • 6–30
pixel order item in • 6–31
pixels per line item in • 6–30
pixel stride item in • 6–30
plane bits per pixel item in • 6–31
Image data unit aggregate (cont'd.)
plane data item in • 6–31
private coding attributes item in • 6–30
scanline stride item in • 6–30
Incremental processing • 4–12 to 4–13
Input formats • 2–9 to 2–10
INSERT AGGREGATE routine • 4–5, CDA–91
Item
accessing • 4–5 to 4–9
array-valued • CDA–80
data types for • 4–6t
definition of • 4–1
determining the address of • 4–9
determining the number of elements in • 4–9
erasing • CDA–68
erasing the contents of • 4–9
finding definition of • CDA–70
locating • CDA–96
writing the contents of • 4–9, CDA–131
Item change list • 6–65

K

Kerning
definition of • 6–52

L

Languages
specifying for processing • 6–5
Latin1 text content • 6–10
Latin1 text content aggregate • 6–10
See also DDIF$_TXT aggregate
Layout • 1–4, 3–14 to 3–17, 6–41
definition of • 3–14
forcing new line, galley, or page through • 6–88
frame-based • 6–41
galley-based • 6–41, 6–42
path-based • 6–41, 6–42
positional • 6–41
position-relative • 6–45
selecting new galley for • 6–89
specifying amount of space after a segment in • 6–90
specifying amount of space before a segment in • 6–90
specifying indentation distance in • 6–89
specifying in-segment break condition in • 6–89

Layout (cont'd.)
specifying leading space between lines in • 6–90
specifying new left indent in • 6–89
specifying new right indent in • 6–90
specifying post-segment break condition in • 6–89
specifying pre-segment break condition in • 6–89
specifying tab stops in • 6–91
text position • 6–46
Layout attribute • 6–9
Layout attributes aggregate • 6–88 to 6–91
See also DDIF$_LL1 aggregate
galley selection item in • 6–89
initial directive item in • 6–88
initial indent indicator item in • 6–89
in-segment break condition item in • 6–89
items in • 6–91t, D–16t
leading ratio item in • 6–90
left indent indicator item in • 6–89
post-segment break condition item in • 6–89
pre-segment break condition item in • 6–89
right indent indicator item in • 6–90
space-after indicator item in • 6–90
space-before indicator item in • 6–90
tab stops item in • 6–91
Layout galley
specifying bounding box information for • 6–26
specifying content streams for • 6–27
specifying flag parameters for • 6–26
specifying outline path for content in • 6–26
specifying reference label for • 6–26
specifying text overflow galley type in • 6–27
Layout galley aggregate • 6–25 to 6–28
See also DDIF$_GLY aggregate
bounding box items for • 6–26
flags item in • 6–26
galley label item in • 6–26
galley outline item in • 6–26
galley streams item in • 6–27
galley successor item in • 6–27
items in • 6–27t, D–6t
LayoutGalley type • E–39
LayoutPrimitive type • E–39
LEAVE SCOPE routine • 4–12, CDA–94
Legend
See Content
Legend attributes • 6–40
Line
specifying attributes for • 6–52 to 6–54
specifying denominator of miter ratio of • 6–54
specifying ending shape of • 6–53
specifying ending size of • 6–54
Line (cont'd.)
specifying mask pattern of • 6–53
specifying numerator of miter ratio of • 6–54
specifying pattern for • 6–52
specifying pattern size of • 6–52
specifying shape of joins of • 6–54
specifying shape of the endings of • 6–53
specifying width of • 6–52
Line-style definition
specifying line-style pattern in • 6–71
specifying private data for • 6–72
specifying reference number for • 6–71
Line-style definition aggregate • 6–71 to 6–72
See also DDIF$_LSD aggregate
items in • 6–72t, D–12t
line-style number item in • 6–71
line-style pattern item in • 6–71
line-style private data item in • 6–72
LOCATE ITEM routine • 4–9, CDA–96
Lookup table entry aggregate • 6–82
See also DDIF$_RGB aggregate
blue value item in • 6–82
green value item in • 6–82
index item in • 6–82
items in • 6–82t, D–15t
red value item in • 6–82

M

Marker
specifying attributes for • 6–55
specifying pattern for • 6–55
specifying size for • 6–55
specifying symbol used as • 6–55
Markup system • 1–6
Measurement enumeration • 4–7
Memory management routines • CDA–40, CDA–44, CDA–48, CDA–110, CDA–113

N

NEXT AGGREGATE routine • 4–5, CDA–99

O

Object identifier

Object identifier (cont'd.)
translating to root aggregate • 4–4
OBJECT ID TO AGGREGATE TYPE routine • 4–4, CDA–101
Occurrence definition
specifying permitted types of • 6–80
specifying structure definition in • 6–80
Occurrence definition aggregate • 6–80 to 6–81
See also DDIF$_OCC aggregate
items in • 6–81t, D–14t
occurrence indicator item in • 6–80
structure element indicator item in • 6–80
OPEN CONVERTER routine • 4–13, CDA–103
OPEN FILE routine • 4–2, CDA–106
OPEN STREAM routine • 4–3, CDA–112
OPEN TEXT FILE routine • 4–3, CDA–116
Options file • 2–4
Output formats • 2–10 to 2–15

P

Page description
including private data in • 6–93
specifying reference label for • 6–93
specifying the type of • 6–93
Page description aggregate • 6–93 to 6–94
See also DDIF$_PGD aggregate
indicator item in • 6–93
items in • 6–93t, D–17t
label item in • 6–93
private data item in • 6–93
Page layout
specifying frame for • 6–95
specifying nominal measure for • 6–94
specifying orientation of • 6–95
specifying prototype for • 6–95
specifying reference label for • 6–94
specifying *x* shrink amount for • 6–94
specifying *x* stretch amount for • 6–94
specifying *y* nominal measurement for • 6–94
specifying *y* shrink amount for • 6–95
specifying *y* stretch amount for • 6–94
Page layout aggregate • 6–94 to 6–96
See also DDIF$_PGL aggregate
content item in • 6–95
items in • 6–95t, D–17t
layout identifier item in • 6–94
nominal measure indicator item in • 6–94
orientation item in • 6–95
Page layout aggregate (cont'd.)
prototype item in • 6–95
x shrink indicator item in • 6–94
x stretch indicator item in • 6–94
y shrink indicator item in • 6–95
y stretch indicator item in • 6–94
Page selection
specifying page-side criteria for • 6–96
specifying selected layout for • 6–96
Page selection aggregate • 6–96 to 6–97
See also DDIF$_PGS aggregate
items in • 6–97t, D–18t
page-side criteria item in • 6–96
select page layout indicator item in • 6–96
Page set • 3–14
Path-based layout • 6–41, 6–42
Path definition
specifying composite path in • 6–72
specifying private data for • 6–72
specifying reference number for • 6–72
Path definition aggregate • 6–72 to 6–73
See also DDIF$_PHD aggregate
description item in • 6–72
items in • 6–72t, D–13t
number item in • 6–72
private data item in • 6–72
Pattern definition
selecting as either solid color or standard pattern • 6–73
selecting color type for • 6–73
specifying blue intensity for • 6–74
specifying color map for • 6–74
specifying green intensity for • 6–74
specifying image data unit for • 6–74
specifying private data for • 6–74
specifying red intensity for • 6–73
specifying reference number for • 6–73
specifying standard pattern number for • 6–74
Pattern definition aggregate • 6–73 to 6–75
See also DDIF$_PTD aggregate
blue intensity item in • 6–74
colors item in • 6–74
definition indicator item in • 6–73
green intensity item in • 6–74
items in • 6–74t, D–13t
number item in • 6–73
private data item in • 6–74
raster-pattern item in • 6–74
red intensity item in • 6–73
solid color indicator item in • 6–73
standard pattern number item in • 6–74

Polyline
controlling the drawing of line segments of • 6–17
controlling the rendition of • 6–16
specifying the layout of • 6–17
Polyline content aggregate • 6–16 to 6–18
See also DDIF$_LIN aggregate
draw pattern item in • 6–17
flags item in • 6–16
items in • 6–18t, D–4t
line path indicator item • 6–17
Positional layout • 6–41
Position-relative layout • 6–45
PostScript back end • 2–11 to 2–15
data loss in • 2–12
data mapping in • 2–11
processing options in • 2–12
Private content aggregate • 6–24 to 6–25
See also DDIF$_PVT aggregate
external reference index item in • 6–25
items in • 6–25t, D–6t
value indicator item in • 6–24
value name item in • 6–24
Private data • 3–6, 6–24
examples of • 6–24
uses of • 6–24
Private item list • 2–6
Processing options
in PostScript back end • 2–12
in Text back end • 2–11
PRUNE AGGREGATE routine • 4–5, CDA–119
PRUNE POSITION routine • 4–10, CDA–121
PUT AGGREGATE routine • 4–11, CDA–123
PUT DOCUMENT routine • 4–10, CDA–126
Put routine • 5–17, CDA–48

R

Raster image content
See Image content
READ TEXT FILE routine • 4–3, CDA–128
Record definition
specifying segments creating instances of • 6–81
specifying type identifier of • 6–81
specifying variables of • 6–81
Record definition aggregate • 6–81 to 6–82
See also DDIF$_RCD aggregate
contents item in • 6–81
items in • 6–82t, D–14t
tag item in • 6–81
Record definition aggregate (cont'd.)
type item in • 6–81
Reference
processing external • CDA–103
REMOVE AGGREGATE routine • 4–5, CDA–130
Restricted content • 3–6, 6–22 to 6–25
external • 6–22 to 6–23
describing data value of • 6–23
identifying data type of • 6–22
indicating encoding of • 6–23
specifying encoding length of • 6–23
private • 6–24 to 6–25
identifying value of • 6–24
indicating type of data in • 6–24
specifying external reference index for • 6–25
Revisable document • 1–1
relationships in • 3–6 to 3–7
Root aggregate • 6–2
creating • 4–4, CDA–42
definition of • 4–1
deleting • 4–4, CDA–56
document content item in • 6–2
document descriptor item in • 6–2
document header item in • 6–2
items in • 6–2t, D–1t
translating to object identifier • 4–4, CDA–3
Root segment • 3–4, 6–6
Routines
CDA$AGGREGATE_TYPE_TO_OBJECT_ID • 4–4, CDA–3
CDA$CLOSE_FILE • 4–2, CDA–5
CDA$CLOSE_STREAM • 4–3, CDA–7
CDA$CLOSE_TEXT_FILE • 4–3, CDA–8
CDA$CONVERT • CDA–9
CDA$CONVERT_AGGREGATE • 4–10, CDA–26
CDA$CONVERT_DOCUMENT • 4–10, CDA–29
CDA$CONVERT_POSITION • 4–11, CDA–31
CDA$COPY_AGGREGATE • 4–5, CDA–33
CDA$CREATE_AGGREGATE • 4–4, CDA–35
CDA$CREATE_FILE • 4–2, CDA–37
CDA$CREATE_ROOT_AGGREGATE • 4–4, CDA–42
CDA$CREATE_STREAM • 4–3, CDA–46
CDA$CREATE_TEXT_FILE • 4–3, CDA–51
CDA$DELETE_AGGREGATE • 4–5, CDA–54
CDA$DELETE_ROOT_AGGREGATE • 4–4, CDA–56
CDA$ENTER_SCOPE • 4–12, CDA–57
CDA$ERASE_ITEM • 4–9, CDA–68
CDA$FIND_DEFINITION • CDA–70
CDA$FIND_TRANSFORMATION • CDA–73
CDA$FLUSH_STREAM • 4–3, CDA–75

Routines (cont'd.)
CDA$GET_AGGREGATE • 4–11, CDA–77
CDA$GET_ARRAY_SIZE • 4–9, CDA–80
CDA$GET_DOCUMENT • 4–10, CDA–82
CDA$GET_EXTERNAL_ENCODING • CDA–84
CDA$GET_STREAM_POSITION • CDA–86
CDA$GET_TEXT_POSITION • CDA–89
CDA$INSERT_AGGREGATE • 4–5, CDA–91
CDA$LEAVE_SCOPE • 4–12, CDA–94
CDA$LOCATE_ITEM • 4–9, CDA–96
CDA$NEXT_AGGREGATE • 4–5, CDA–99
CDA$OBJECT_ID_TO_AGGREGATE_TYPE • CDA–101
CDA$OPEN_CONVERTER • 4–13, CDA–103
CDA$OPEN_FILE • 4–2, CDA–106
CDA$OPEN_STREAM • 4–3, CDA–112
CDA$OPEN_TEXT_FILE • 4–3, CDA–116
CDA$PRUNE_AGGREGATE • 4–5, CDA–119
CDA$PRUNE_POSITION • 4–10, CDA–121
CDA$PUT_AGGREGATE • 4–11, CDA–123
CDA$PUT_DOCUMENT • 4–10, CDA–126
CDA$READ_TEXT_FILE • 4–3, CDA–128
CDA$REMOVE_AGGREGATE • 4–5, CDA–130
CDA$STORE_ITEM • 4–9, CDA–131
CDA$WRITE_TEXT_FILE • 4–3, CDA–137

S

Scope • 4–12 to 4–13
Segment • 3–2 to 3–6
binding attributes to • 6–7
definition of • 4–1
identifying changed attributes in • 6–65
indicating category of the content of • 6–36
listing the variables bound to • 6–36
referencing a type definition for • 6–6
referencing generic content from • 3–9
referencing generic type from • 3–9
root • 6–6
specifying a reference label for • 6–6
specifying available content definitions for • 6–47
specifying available font definitions for • 6–47
specifying available line style definitions for • 6–47
specifying available path definitions for • 6–47
specifying available pattern definitions for • 6–47
specifying available type definitions for • 6–48
specifying content of • 6–7
specifying content streams for • 6–36
specifying generic layout for • 6–7
specifying language for • 6–39
Segment (cont'd.)
specifying name for • 6–6
specifying private attributes for • 6–35
specifying processing characteristics for • 6–36
specifying specific layout for • 6–7
specifying the type of computed content in • 6–37
using generic content with • 3–8 to 3–9
using generic types with • 3–8
Segment attributes aggregate • 6–35 to 6–69
See also DDIF$_SGA aggregate
alternate presentation item in • 6–41
bits per component item in • 6–60
brightness polarity item in • 6–56
component space organization item in • 6–58
component wavelength indicator item in • 6–58
computed content indicator item in • 6–37
content category item in • 6–36
content definition item in • 6–47
content streams item in • 6–36
cross-reference index item in • 6–38
cross-reference segment label item in • 6–38
cross-reference variable label item in • 6–38
data plane significance item in • 6–59
data-planes-per-pixel item in • 6–59
fixed frame position items in • 6–63
font definition item in • 6–47
frame bounding box items • 6–61
frame clipping path item in • 6–62
frame content transformation item in • 6–65
frame flags item in • 6–60
frame outline item in • 6–61
frame position item in • 6–62
function name item in • 6–38
function parameters item in • 6–39
galley frame items in • 6–63 to 6–64
galley layout item in • 6–42
grid type item in • 6–56
horizontal alignment item in • 6–44
inline frame items in • 6–63
item change list item in • 6–65
items in • 6–66t, D–8t
language item in • 6–39
layout format item in • 6–42
layout indicator item • 6–41
layout path item in • 6–42
legend unit denominator item in • 6–40
legend unit name item in • 6–40
legend unit numerator item in • 6–40
line end finish item in • 6–53
line end size indicator item in • 6–54
line end start item in • 6–53

- Segment attributes aggregate (cont'd.)
  - line interior pattern item in • 6–54
  - line joint item in • 6–54
  - line mask pattern item in • 6–53
  - line pattern size item in • 6–52
  - line progression item in • 6–56
  - line progression path aspect ratio item in • 6–56
  - line style definition item in • 6–47
  - line style item in • 6–52
  - line width indicator item in • 6–52
  - lookup table item in • 6–57
  - margin frame items in • 6–64 to 6–65
  - marker mask pattern item in • 6–55
  - marker size indicator item in • 6–55
  - marker style item in • 6–55
  - miter limit denominator item in • 6–54
  - miter limit numerator item in • 6–54
  - number of components item in • 6–60
  - path definition item in • 6–47
  - path orientation indicator item • 6–43
  - pattern definition item in • 6–47
  - pixel path aspect ratio item in • 6–56
  - pixel path item in • 6–56
  - private attributes item in • 6–35
  - private data item in • 6–56
  - reference index item in • 6–38
  - reference target item in • 6–38
  - relative horizontal character position item in • 6–45
  - relative vertical character position item in • 6–45
  - segment binding item in • 6–36
  - segment tags item in • 6–36
  - spectral component mapping item in • 6–57
  - structure attributes items in • 6–39
  - text character decimal alignment item in • 6–51
  - text direction item in • 6–51
  - text font item in • 6–48
  - text kerning item in • 6–52
  - text leader attribute items in • 6–51 to 6–52
  - text mask pattern item in • 6–48
  - text position indicator item in • 6–46
  - text rendition item in • 6–49
  - text size attribute items in • 6–50
  - timing descriptor item in • 6–57
  - type definition item in • 6–48
  - unit name item in • 6–40
  - units per measurement item in • 6–40
  - variable item in • 6–38
  - vertical alignment item in • 6–44
  - wrap attributes item in • 6–42
- Segment binding
  - specifying computed variable items in • 6–77
  - specifying counter variable items for • 6–76 to 6–77
  - specifying list variable items in • 6–77
  - specifying name of variable being defined in • 6–75
  - specifying type of variable value in • 6–75
- Segment binding aggregate • 6–75 to 6–78
  - See also DDIF$_SGB aggregate
  - computed variable items in • 6–77
  - counter variable items in • 6–76 to 6–77
  - items in • 6–78t, D–13t
  - list variable items in • 6–77
  - variable name item in • 6–75
  - variable value indicator item in • 6–75
- Segment tag • 6–9
  - private • 6–9
  - standard • 6–9
- Sequence
  - definition of • 4–1
  - inserting an aggregate into • 4–5
  - locating next aggregate in • 4–5
  - removing an aggregate from • 4–5, CDA–130
- Soft content • 3–1
- Soft directive • 6–11
  - values for • 6–12t
- Soft directive aggregate • 6–11
  - DDIF$_SFT aggregate
  - soft directive item in • 6–11
- Soft value directive • 6–14 to 6–15
  - specifying escapement constant for • 6–14
  - specifying escapement ratio denominator for • 6–14
  - specifying escapement ratio numerator for • 6–14
  - specifying new variable value for • 6–15
  - specifying type of • 6–14
  - specifying variable to be reset by • 6–15
- Soft value directive aggregate • 6–14 to 6–15
  - See also DDIF$_SFV aggregate
  - directive choice item in • 6–14
  - escapement constant indicator in • 6–14
  - escapement ratio item in • 6–14
  - items in • 6–15t, D–3t
  - reset value item in • 6–15
  - reset variable item in • 6–15
- Specific attribute
  - precedence of • 3–9
- Specific layout
  - specifying type of layout for • 6–85
- Specific layout aggregate • 6–85 to 6–86

Specific layout aggregate (cont'd.)
See also DDIF$_LS1 aggregate
items in • 6–85t, D–15t
layout indicator item in • 6–85
Still image • 6–21
Stored semantics file attribute • A–1
See also File tag
STORE ITEM routine • 4–9, CDA–131
Stream
closing • 4–2, 4–3, CDA–5, CDA–7
creating • 4–2, 4–3, CDA–37, CDA–46, CDA–106
definition of • 4–1
flushing contents of • 4–3, CDA–75
opening • 4–2, CDA–112
retrieving position in • CDA–86
retrieving size of • CDA–86
returning position in • CDA–31
returning size of • CDA–31
writing a document to • CDA–126
writing aggregates to • CDA–123
Structure attributes
specifying legal types of • 6–39
Structured document • 1–4
Subaggregate
definition of • 4–4
Syntax diagrams
Angle • E–25
AngleRef • E–25
Arc • E–12
ArcPath • E–30
ASCIIString • E–24
BeginSegment • E–6
Binding • E–35
BoundingBox • E–18
BreakCriteria • E–41
Category Tag • E–34
Color • E–18
CompositePath • E–29
ComputeDefn • E–19
Conformance Tag • E–34
ContentDefn • E–23
ContentReference • E–18
ContentReferencePrimitive • E–17
CounterDefn • E–35
CounterStyle • E–36
CrossRef • E–19
CubicBezier • E–12
CubicBezierPath • E–30
DDIFDocument • E–4
Directive • E–10
DocumentDescriptor • E–5
Syntax diagrams (cont'd.)
DocumentHeader • E–5
Document root segment • E–5
Escapement • E–19
EscapementDirective • E–10
Expression • E–36
ExternalReference • E–20
ExternalRefIndex • E–23
FillAreaSet • E–12
FontDefn • E–20
FontNumber • E–28
Format • E–21
FormattingPrimitive • E–9
FrameParameters • E–21
FunctionLink • E–23
GalleyAttributes • E–40
GalleyFrameParams • E–21
GalleyVerticalPosition • E–22
GenericLayout • E–38
GenMeasure • E–42
GenSize • E–42
GraphicsPrimitive • E–11
ImageAttributes • E–15
ImageCodingAttrs • E–15
ImagePrimitive • E–14
ImgCmptSpcAttrs • E–17
ImgLutData • E–16
InlineFrameParams • E–21
Label • E–24
Label types • E–24
LanguageIndex • E–23
LayoutAttributes • E–41
LayoutGalley • E–39
LayoutObjectType • E–36
LayoutPrimitive • E–39
LeaderStyle • E–8
LegendUnits • E–25
LineAttributes • E–12
LineDefn • E–30
LineEndNumber • E–13
LineJoin • E–14
LineStyleNumber • E–13
MarginFrameParams • E–22
MarginHorizontalPosition • E–22
MarkerAttributes • E–14
MarkerNumber • E–14, E–28
Measure • E–25
MeasurementUnits • E–27
NamedValue • E–27
NamedValueList • E–28
NamedValueTag • E–34
OccurrenceDefn • E–33

Syntax diagrams (cont'd.)
PageDescription • E–38
PageLayout • E–39
PageSet • E–39
PathDefn • E–29
PathNumber • E–28
PatternDefn • E–31
PatternNumber • E–29
Polyline • E–11
PolyLinePath • E–30
Position • E–26
Ratio • E–26
RecordDefn • E–37
RecordList • E–37
Reference • E–31
RenditionCode • E–7
RestrictedContent • E–17
RGB • E–18
RightAngle • E–26
SegmentAttributes • E–31
SegmentPrimitive • E–6
SegmentTag • E–34
SegTypeDefn • E–32
Size • E–26
SpecificLayout • E–40
StandardPattern • E–31
StorageSystemTag • E–34
StreamTag • E–35
StringExpression • E–37
StringLayout • E–8
StructureDefinition • E–32
StructureElement • E–33
TabStop • E–43
TabStopList • E–42
Tag • E–33
TextAttributes • E–7
TextLayout • E–8
TextPrimitive • E–6
Transformation • E–35
ValueData • E–28
ValueDirective • E–9
VariableLabel • E–24
VariableReset • E–10
WrapAttributes • E–40
XCoordinate • E–26
YCoordinate • E–27

T

Tab stop
Tab stop (cont'd.)
specifying horizontal position of • 6–97
specifying leader character for • 6–98
specifying type of alignment for • 6–97
Tab stop aggregate • 6–97 to 6–98
See also DDIF$_TBS aggregate
items in • 6–98t, D–18t
tab stop horizontal position indicator item in • 6–97
tab stop leader item in • 6–98
tab stop type item in • 6–97
Tag
See File tag
Text
controlling kerning for • 6–52
specifying alignment characters for • 6–51
specifying amount of space used by the leader character in • 6–51
specifying attributes for • 6–48 to 6–52
specifying format of lines wrapped by the formatter • 6–86
specifying format of lines wrapped by the user • 6–86
specifying leader alignment in • 6–51
specifying maximum consecutive hyphenated lines of • 6–87
specifying maximum orphan size of • 6–87
specifying maximum widow size of • 6–87
specifying one or more renditions for • 6–49
specifying pattern and color of glyphs in • 6–48
specifying rules that affect hyphenation of • 6–87
specifying string used to fill leader space in • 6–51
specifying the direction of characters in • 6–51
specifying the font used for • 6–48
specifying the height of • 6–50
specifying the ratio for character widths in • 6–50
specifying type of leader to use in • 6–51
Text attribute • 6–9, 6–48
Text back end • 2–11
data loss in • 2–11
data mapping in • 2–11
processing options in • 2–11
Text content • 3–1, 3–5, 6–9
general • 6–10
Latin1 • 6–10
specifying general character set for • 6–10
specifying Latin1 character set for • 6–10
Text content aggregate
See also DDIF$_TXT aggregate
item in • 6–10

Text file
closing • 4–3, CDA–8
creating • 4–3, CDA–51
opening • 4–3, CDA–116
reading a line from • 4–3, CDA–128
returning position in • CDA–89
returning size of • CDA–89
writing a line to • 4–3, CDA–137
Text front end • 2–10
data loss in • 2–10
data mapping in • 2–10
document syntax errors in • 2–10
external file references in • 2–10
Text kerning • 6–52
Time-varying image • 6–21
Transformation
returning information about • CDA–73
specifying type of parameter specified for • 6–83
specifying value of the parameter for • 6–84
Transformation aggregate • 6–83 to 6–84
See also DDIF$_TRN aggregate
items in • 6–84t, D–15t
transformation parameter indicator item in • 6–83
Type
generic • 3–8
referencing • 3–9
Type definition
specifying parent for • 6–78
specifying private data for • 6–78
specifying reference label for • 6–78
specifying segment attributes for • 6–78
Type definition aggregate • 6–78 to 6–79
See also DDIF$_TYD aggregate
attributes item in • 6–78
items in • 6–79t, D–14t
label item in • 6–78
parent item in • 6–78
private data item in • 6–78

U

User routine
allocation • CDA–40, CDA–44, CDA–48, CDA–110, CDA–113
deallocation • CDA–40, CDA–44, CDA–48, CDA–110, CDA–113
Flush • CDA–76
Get • 5–13, CDA–114
Get-position • CDA–87
User routine (cont'd.)
Put • CDA–48

V

Variable
encoding of • 4–7
Variable computed content • 6–38
VIEW command • 2–7
VMS
support for CDA in • A–1

W

Wrap attributes aggregate • 6–86 to 6–88
See also DDIF$_LW1 aggregate
hyphenation flags item in • 6–87
hyphenation lines item in • 6–87
items in • 6–88t, D–15t
maximum orphan size item in • 6–87
maximum widow size item in • 6–87
quad format item in • 6–86
wrap format item in • 6–86
WRITE TEXT FILE routine • 4–3, CDA–137

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local DIGITAL subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local DIGITAL subsidiary or approved distributor
Internal[1]	————	SDC Order Processing - WMO/E15 *or* Software Distribution Center Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

VMS Compound Document Architecture Manual
AA–MG30A–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

-- Do Not Tear - Fold Here and Tape ----------

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here ----------

Reader's Comments

VMS Compound Document Architecture Manual
AA–MG30A–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

-— Do Not Tear - Fold Here and Tape ----------

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-— Do Not Tear - Fold Here ----------