# CDA Reference Manual: Volume 2

Order Number: Volume 2: AA–PC02A–TE

**June 1990**

This manual provides reference material for using the CDA Toolkit to create compound document applications, converters, and viewers.

**Revision/Update Information:** This revised manual supersedes the *CDA Reference Manual* Part I (Order No. AA–PABUA–TE) and Part II (Order No. AA–PABVA–TE).

**Software Version:** VMS Version 5.4
ULTRIX Version 4.0

**digital equipment corporation**
**maynard, massachusetts**

**First Printing, December 1988**
**Revised, October 1989**
**Revised, June 1990**



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



ZK5428

This document was prepared using VAX DOCUMENT, Version 1.2

# Contents

## Chapter 3 Transferring CDA Documents

## Chapter 4 DDIF Structures

## Chapter 8 CDA Toolkit Routines

## Examples



## Figures

## Tables

# Preface

This manual provides reference material for using the CDA (Compound Document Architecture) Toolkit to create compound document applications, converters, and viewer widgets. Information in this manual includes reference material for using the DDIF (DIGITAL Document Interchange Format) and DTIF (DIGITAL Table Interchange Format) aggregates that are processed by the CDA Toolkit routines.

The CDA Toolkit is a collection of data structures and routines that support the creation of CDA applications. The CDA Converter architecture is used to convert files of a specified input format to a specified output format. The CDA Viewer is used to display CDA-encoded files on a workstation display or character cell terminal.

CDA is supported in both the VMS and the ULTRIX environments. The information contained in this manual is appropriate for both systems. Any differences between the two implementations are called out in the text of this manual.

All of the following products support CDA-encoded files. If you intend to manipulate only 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 |
| DEC GKS | DEC GKS-3D | PHIGS |
| CDA Viewer | DECwindows MAIL | DECImage Applications Services |
| Converters | MAIL | DECwrite |
| DECchart | DECdecision | DEC Test Manager |

## Intended Audience

This manual is intended for system and application programmers who already have been introduced to CDA and who are ready to use the CDA Toolkit to write compound document applications, converters, or viewers. Some knowledge of the tasks and terminology associated with document typesetting is helpful.

## Document Structure

This manual consists of 13 chapters, several appendixes, and a glossary, as follows:

- Chapter 1, Introduction provides an introduction to the reference material describing the aggregates and routines contained in the CDA Toolkit.

- Chapter 2, Bundled Converter Formats and Commands describes the VMS and ULTRIX converter formats and commands used to convert and to view CDA documents.
- Chapter 3, Transferring CDA Documents describes how to mail and to copy CDA documents on VMS and ULTRIX systems.
- Chapter 4, DDIF Structures describes each of the DDIF aggregate structures.
- Chapter 5, DTIF Structures describes each of the DTIF aggregate structures.
- Chapter 6, CFE Structures describes each of the CFE aggregate structures.
- Chapter 7, ESF Structures describes each of the ESF aggregate structures.
- Chapter 8, CDA Toolkit Routines describes each of the routines contained in the CDA Toolkit. 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.
- Chapter 9, User-Defined Routines describes the user-defined routines used to write CDA-conforming applications and front and back ends.
- Chapter 10, CDA Toolkit Example Program 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.
- Chapter 11, CDA Converter Routines describes each of the converter routines that must be created in order to write a CDA-conforming front or back end.
- Chapter 12, Text Front End Source File 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.
- Chapter 13, CDA Viewer Routines describes each of the viewer routines used to create a character-cell or DECwindows viewer application.
- Appendix A, DDIF Fill Patterns illustrates the CDA-defined fill patterns.
- Appendix B, DDIF Syntax Diagrams contains a brief overview of DDIS (DIGITAL Data Interchange Syntax) followed by the syntax diagrams for the various constructs supported by the DDIF architecture.
- Appendix C, DTIF Syntax Diagrams contains the syntax diagrams for the various constructs supported by DTIF.
- Appendix D, CFE Syntax Diagrams contains the syntax diagrams for the various constructs supported by CFE.
- Appendix E, ESF Syntax Diagrams contains the syntax diagrams for the various constructs supported by ESF.
- Appendix F, VMS Support for CDA in DECwindows discusses the support provided by VMS for the CDA Toolkit and the tagging of DDIF-encoded files.
- Appendix G, CDA$ Facility Messages lists and describes the CDA$_ facility messages generated by the CDA Toolkit.
- Glossary, Glossary of Terms defines the terminology associated with the CDA Toolkit and CDA Converter Architecture.

## Associated Documents

CDA 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.

The complete CDA documentation set includes two tutorials and a reference manual:

- *Introduction to the CDA Services*
- *Guide to Creating Compound Documents with the CDA Toolkit*
- *CDA Reference Manual*

The CDA documentation set is a separately orderable subkit available for purchase with the VMS and ULTRIX operating system documentation. Each manual in the CDA documentation set is also available for separate purchase.

The CDA Converter Library end-user documentation set describes additional document, graphics, image, and table data file formats that are supported by the CDA Converter architecture, but that are not bundled with the VMS or ULTRIX operating system. The following two manuals describe the additional interchange formats:

- *Guide to the CDA Converter Library*
- *Getting Started with the CDA Converter Library*

## Conventions

The following conventions are used in this manual:

| | |
|---|---|
| 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:<br>• Additional optional arguments in a statement have been omitted.<br>• The preceding item or items can be repeated one or more times.<br>• Additional parameters, values, or other information can be entered. |
| .<br>.<br>. | 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. For online manuals, user input is specified in **bold**. |
| *italic text* | *Italic text* represents the introduction of a new term or the name of an argument, an attribute, or a reason. |
| *italic text* | *Italic text* represents user-written routines (for example, *get-aggregate*). |
| **boldface text** | **Boldface text** represents information that can vary in system messages (for example, Internal error **number**). |
| 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. |
| lowercase text | Lowercase letters indicate the names of the CDA Toolkit VAX format routines and values that are portable to ULTRIX systems. Value names that appear in lowercase must be coded as such in order to be portable to ULTRIX systems. |
| | 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 radixes—binary, octal, or hexadecimal—are explicitly indicated in the coding examples. |

Chapter 8

# CDA Toolkit Routines

This section describes the CDA Toolkit routines and VMS and ULTRIX compile and link procedures used to create CDA-conforming applications. Each routine description includes the following information:

- A common language (VAX) style binding that is supported on both VMS and ULTRIX systems
- An ULTRIX C style binding that is supported on both VMS and ULTRIX systems
- A description of the value returned by the routine
- Descriptions of each routine argument
- A description of the routine itself
- A list of possible values returned by the routine

**NOTE**

Given the bindings available at this time, there are two ways to create CDA Toolkit applications that are portable across VMS and ULTRIX systems:

1. Code using C style bindings by matching mix-cased spelling.
2. Code your source in lowercase if using the VAX formats (as the bindings are lowercase in the current ULTRIX CDA Toolkit).

If you are programming in Ada, please refer to the *Guide to Applications Programming* for information on Ada programming with DECwindows.

## 8.1 Compile and Link Procedures for Applications

To create a VMS or ULTRIX application using the CDA Toolkit routines, include the following public files in your source code:

| VMS and ULTRIX File Names | Description |
|---|---|
| SYS$LIBRARY:cda$def.h<br>/usr/include/cda_def.h | CDA Toolkit keyword definitions |
| SYS$LIBRARY:ddif$def.h<br>/usr/include/ddif_def.h | DDIF aggregate definitions |

| VMS and ULTRIX File Names | Description |
|---|---|
| SYS$LIBRARY:dtif$def.h<br>/usr/include/dtif_def.h | DTIF aggregate definitions |
| SYS$LIBRARY:cda$msg.h<br>/usr/include/cda_msg.h | CDA error messages |

**NOTE**

> If you are programming in Ada, please refer to the *Guide to Applications Programming* for information on Ada programming with DECwindows.

Section 8.1.1 describes the VMS compile and link procedure for CDA applications. Section 8.1.2 describes the ULTRIX compile and link procedure for CDA applications.

### 8.1.1 VMS Link Procedure

You can compile and link a CDA application on VMS using the following build procedure that also incorporates debugging:

```
$ CC /DEBUG MY APPLICATION
$ LINK /DEBUG MY_APPLICATION,SYS$INPUT:/OPTION
SYS$LIBRARY:CDA$ACCESS/SHARE
```

### 8.1.2 ULTRIX Link Procedure

You can compile and link an application on ULTRIX using the following build procedure:

```
% cc -o my_application my_application.c -lddif -lm
```

The **-lm** parameter specifies the math library that is required by the CDA Toolkit routines (**-lddif**).

# AGGREGATE TYPE TO OBJECT ID

Translates a root aggregate type to an object identifier.

## VAX FORMAT

**status = cda$aggregate_type_to_object_id**

*(aggregate-type ,buf-len ,buf-adr ,nam-len ,nam-buf ,act-nam-len ,act-len)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **aggregate-type** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **buf-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **buf-adr** | VMS usage: | array |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference, array reference |
| **nam-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

| Argument | Argument Information | |
|---|---|---|
| **nam-buf** | VMS usage: | array |
| | Data type: | character string |
| | Access: | write only |
| | Mechanism: | by reference, array reference |
| **act-nam-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **act-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaAggregateTypeToObjectId**
*(aggregate_type, buf_len, buf_adr, nam_len, nam_buf, act_nam_len, act_len)*

## Argument Information

```
unsigned long CdaAggregateTypeToObjectId(aggregate_type,
                        buf_len, buf_adr, nam_len,
                        nam_buf, act_nam_len, act_len)
              unsigned long      aggregate_type;
              unsigned long      buf_len;
              unsigned long      buf_adr[];
              unsigned long      nam_len;
              unsigned char      nam_buf[];
              unsigned long      *act_nam_len;
              unsigned long      *act_len;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***aggregate-type***
Type of the aggregate being translated. The root aggregate type must be either DDIF$_DDF or DTIF$_DTF.

***buf-len***
Length (in bytes) of the object identifier buffer. Length must be at least 28 bytes (space for 7 longwords).

***buf-adr***
An array of longwords to receive the object identifier.

***nam-len***
Length (in bytes) of the domain name buffer.

***nam-buf***
Address of the domain name buffer to receive a string of characters.

***act-nam-len***
Receives the actual length (in bytes) of the domain name in the **nam-buf** buffer.

***act-len***
Receives the actual length (in bytes) of the object identifier.

## Description

The AGGREGATE TYPE TO OBJECT ID routine translates a root aggregate type to an object identifier.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_INVAGGTYP | Invalid aggregate type |
| CDA$_INVBUFLEN | Invalid buffer length |

# CLOSE FILE

Closes the specified compound document file and stream. If the file being closed was receiving output, the CLOSE FILE routine writes any buffered data before closing the file and stream.

## VAX FORMAT

**status = cda$close_file**

*(stream-handle ,file-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **stream-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **file-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaCloseFile**

*(stream_handle, file_handle)*

## Argument Information

```
unsigned long CdaCloseFile(stream_handle,
                 file_handle)
          unsigned long       stream_handle;
          unsigned long       file_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***stream-handle***
Handle of the stream to be closed. This handle is returned by a call to either the OPEN FILE routine or the CREATE FILE routine.

***file-handle***
Handle of the file to be closed. 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. If the file being closed was receiving output, this routine writes out any buffered data before closing the stream. Note that the **stream-handle** and **file-handle** values are invalid after a call to this routine.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

Any error returned by the memory deallocation routines.

Any error returned by the file routines.

## Example

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.

```
    .
    .
    .
/* 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);

    .
    .
    .
```

# CLOSE STREAM

Closes an open compound document stream.

## VAX FORMAT

**status = cda$close_stream**

*(stream-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **stream-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaCloseStream**

*(stream_handle)*

## Argument Information

```
unsigned long CdaCloseStream(stream_handle)
               unsigned long       stream_handle;
```

# CLOSE STREAM

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Argument

***stream-handle***
Handle of the stream to be closed. 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. If the stream being closed was receiving output, 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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

Any error returned by the memory deallocation routines.

# CLOSE TEXT FILE

Closes a text file.

## VAX FORMAT

**status = cda$close_text_file**
*(text-file-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **text-file-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaCloseTextFile**
*(text_file_handle)*

## Argument Information

```
unsigned long CdaCloseTextFile(text_file_handle)
               unsigned long     text_file_handle;
```

# CLOSE TEXT FILE

## RETURNS

***status***
A condition value indicating the return status of the routine call

## Argument

***text-file-handle***
Identifier of the text file to be closed. 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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

Any error returned by the memory deallocation routines.

Any error returned by the file routines.

# CONVERT

Lets the user perform document conversion from within an application. This includes beginning, continuing, or discontinuing the conversion of a document.

## VAX FORMAT

**status = cda$convert**

*(function-code ,standard-item-list ,private-item-list ,converter-context)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **function-code** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **standard-item-list** | VMS usage: | item_list_2 |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference, array reference |
| **private-item-list** | VMS usage: | unspecified |
| | Data type: | unspecified |
| | Access: | read only |
| | Mechanism: | by reference |
| **converter-context** | VMS usage: | context |
| | Data type: | longword (unsigned) |
| | Access: | read only or write only |
| | Mechanism: | by reference |

# CONVERT

## C FORMAT

**status = CdaConvert**

*(function_code, standard_item_list, private_item_list, converter_context)*

## Argument Information

```
unsigned long CdaConvert(function_code, standard_item_list,
                 private_item_list, converter_context)
              unsigned long          function_code;
              unsigned long          *standard_item_list;
              unsigned char          *private_item_list;
              unsigned long          *converter_context;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***function-code***
Symbolic constant that identifies the function to be performed. These symbolic constant values are defined in the file cda$def.h on VMS systems and in the file cda_def.h on ULTRIX systems. 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***
An item list that identifies the document source and destination, and can also contain options to control processing.

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. 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 main entry point for the 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 a routine within 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$_INPUT_FRONT_END_DOMAIN***
The parameter is the address and length of a string that specifies the input document domain (either DDIF or DTIF). This item is used with the CDA$_INPUT_PROCEDURE item to denote the input domain.

***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 will supply an appropriate backup default file specification.

***CDA$_INPUT_PROCEDURE***
The parameter is the address of a procedure to provide input to the front end. 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 Chapter 9.

***CDA$_INPUT_PROCEDURE_PARM***
The parameter is the address of a longword parameter to the input procedure. The item-list length field must be set to 4. This item is valid only if the CDA$_INPUT_PROCEDURE value is specified

***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 position procedure must conform to the requirements for a user *get-position* routine. 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 entire in-memory structure, including 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 main entry point of the 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$_OUTPUT_BACK_END_DOMAIN***
The parameter is the address and length of a string that specifies the output document domain (either DDIF or DTIF). This item is used with the CDA$_OUTPUT_PROCEDURE item to denote the output domain.

***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 will 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 Chapter 9.

***CDA$_OUTPUT_PROCEDURE_PARM***
The parameter is the address of a longword parameter to the output procedure. The item-list length field must be set to 4. This item is valid only if the CDA$_OUTPUT_PROCEDURE item is specified.

***CDA$_OUTPUT_PROCEDURE_BUFFER***
The parameter is the address and length of the initial output buffer for the output procedure. This item is valid only if the CDA$_OUTPUT_PROCEDURE item is specified.

***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. This root aggregate contains an entire in-memory document at the end of the conversion.

***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 on VMS systems and cda_options on ULTRIX systems. 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.

***CDA$_OPTIONS_LINE***
The parameter is the address and length of a string that contains options to control processing. The format of each string is defined in the description of the CDA$_OPTIONS_FILE item code.

***private-item-list***
A private item list that is passed directly to the output converter module that is invoked. 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. On ULTRIX systems, the CDA$_OUTPUT_BACK_END_PROCEDURE item must be specified when this parameter is used.

***converter-context***
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$_STOP as the function code. This value is invalidated when the CONVERT routine returns a status other than CDA$_SUSPEND. This parameter is used by the back end to store its processing context.

## 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.
- Either 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$_OUTPUT_BACK_END_PROCEDURE item, but not both, can be specified once in the item list. If neither is specified, the default output format is DDIF.
- Either the CDA$_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.
- The CDA$_OPTIONS_LINE item can be specified multiple times in the item list.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_DCVNOTFND | Domain converter not found. |
| CDA$_ICVNOTFND | Input converter not found. |
| CDA$_INVFUNCOD | Invalid function code. |
| CDA$_INVINPDMN | Invalid input domain. |
| CDA$_INVITMLST | Invalid item list. |
| CDA$_INVOUTDMN | Invalid output domain. |
| CDA$_NORMAL | Normal successful completion. |
| CDA$_OCVNOTFND | Output converter not found. |
| CDA$_SUSPEND | Converter is suspended. |
| CDA$_UNSUPCNV | Unsupported conversion. |
| CDA$_UNSUPFMT | Unsupported document format. |

Any error conditions returned by the specific front end.

Any error conditions returned by the specific back end.

## Example

This example illustrates the use of the CONVERT routine to invoke the DDIF and Text converters.

```
/* Example text to ddif conversion using callable converter interface
 * with a user-supplied get-rtn for text input.
 */

#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>
#include <stdio.h>
#endif

#define FAILURE(x)     (((x) & 1) == 0)

#define text_ubf_size 2048
```

```
/* User-supplied get-prm
 */
#ifdef vms
struct FAB      text_fab;
struct RAB      text_rab;
#else
struct urab {
    int fd;                         /* file descriptor     */
    FILE *fs;                       /* file ptr, used for text files */
    unsigned char *fil_buffer;      /* address of input buffer */
    unsigned long fil_buflen;       /* length of input buffer */
    unsigned long fil_size;         /* size of file */
};
struct urab     text_rab;
extern char *fgets();
#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;


/* User-supplied get-rtn
 */
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;
{

#ifdef vms
unsigned long   status;

        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
char *status;
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");
}
```

To compile and run this program on VMS systems, you can use the following DCL commands:

```
$ CC
_$ /OPTIMIZE=NODISJOINT -
_$ /NOLIST -
_$ TEXT_CONVERTER.C

$ LINK /EXE=TEXT_CONVERTER -
_$ /NOMAP -
_$ TEXT_CONVERTER,SYS$INPUT:/OPTION
SYS$LIBRARY:CDA$ACCESS/SHARE
SYS$SHARE:VAXCRTL/SHARE

$ RUN TEXT_CONVERTER
```

To compile and run this program on ULTRIX systems, you can use the following commands:

```
% cc -o text_converter text_converter.c -lddif -lm
% text_converter
```

# CONVERT AGGREGATE

the aggregate type

Reads the next aggregate from a specified front end.

## VAX FORMAT

**status = cda$convert_aggregate**

*(root-aggregate-handle ,front-end-handle ,aggregate-handle ,aggregate-type)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **front-end-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **aggregate-type** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaConvertAggregate**

*(root_aggregate_handle, front_end_handle, aggregate_handle, aggregate_type)*

## Argument Information

```
unsigned long CdaConvertAggregate(root_aggregate_handle,
                  front_end_handle, aggregate_handle,
                  aggregate_type)
              unsigned long     root_aggregate_handle;
              unsigned long     front_end_handle;
              unsigned long     *aggregate_handle;
              unsigned long     *aggregate type;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate associated with the aggregate to be read. 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 the aggregates, you cannot specify the same **root-aggregate-handle** again when calling this routine.

***front-end-handle***
Identifier of the front end that reads the aggregate. This handle is either returned by a call to the OPEN CONVERTER routine, or is passed as a parameter to the ddif$write_**format** or dtif$write_**format** entry point in the back end.

***aggregate-handle***
Receives the handle of the aggregate just read. This handle must be used in all subsequent operations on that aggregate.

***aggregate-type***
Receives the aggregate type. If the aggregate type is DDIF$_EOS (end of segment), then the value of **aggregate-handle** is 0.

The aggregate type returned can be any one of the primary DDIF or DTIF aggregates:

| 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$_EOS | End of segment |
| DTIF$_DSC | Document descriptor |
| DTIF$_HDR | Document header |
| DTIF$_TBL | Table definition |
| DTIF$_ROW | Row definition |
| DTIF$_CLD | Cell data |

Note that the returned aggregate is not part of a sequence.

## Description

The CONVERT AGGREGATE routine lets your application read the next aggregate from the specified front end. This routine can only be invoked by a back end.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_ENDOFDOC | End of document |
| CDA$_INVAGGTYP | Invalid aggregate type |
| CDA$_INVDOC | Invalid document content |
| CDA$_NORMAL | Normal successful completion |
| CDA$_UNSUPCNV | Unsupported conversion |

Any error returned by the memory allocation routines.

Any error returned by the file routines.

# CONVERT DOCUMENT

Reads an entire document from a specified front end.

## VAX FORMAT

**status = cda$convert_document**

*(root-aggregate-handle ,front-end-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **front-end-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaConvertDocument**

*(root_aggregate_handle, front_end_handle)*

## Argument Information

```
unsigned long CdaConvertDocument(root_aggregate_handle,
                                 front_end_handle)
              unsigned long          root_aggregate_handle;
              unsigned long          front_end_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate associated with the document being read. 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***
Identifier of the front end that reads the document. This handle is passed to the back end as a parameter to the ddif$write_**format** or dtif$write_**format** entry point in the back end. In addition, when a front end calls the OPEN CONVERTER routine, the new front end handle is returned.

## Description

The CONVERT DOCUMENT routine lets your application read an entire document from the specified front end. This routine can only be invoked by a back end. On return from this routine, the entire document is present in memory in aggregates linked from the document root aggregate.

## RETURN VALUES

| Return Value | Description |
|---|---|
| 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.

# CONVERT POSITION

Returns the current position in the input stream being processed, as well as the total size of the input stream.

## VAX FORMAT

**status = cda$convert_position**

*(front-end-handle ,stream-position ,stream-size)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **front-end-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **stream-position** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **stream-size** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaConvertPosition**

*(front_end_handle, stream_position, stream_size)*

# CONVERT POSITION

## Argument Information

```
unsigned long CdaConvertPosition(front_end_handle,
                         stream_position, stream_size)
         unsigned long       front_end_handle;
         unsigned long       *stream_position;
         unsigned long       *stream_size;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***front-end-handle***
Identifier of the front end that is processing the input stream. The front end handle is either returned by a call to the OPEN CONVERTER routine or is passed as a parameter to ddif$write_**format** or dtif$write_**format**.

***stream-position***
Receives the current position (in bytes or aggregates) as measured from the start of the input stream being processed.

***stream-size***
Receives the total size (in bytes or aggregates) of the input stream being processed.

## Description

The CONVERT POSITION routine returns the current position in the input stream being processed, as well as the total size of a document being processed by the CONVERT AGGREGATE routine. The numbers are either in units of bytes or aggregates.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

Any condition value returned by the front end *get-position* procedure.

# COPY AGGREGATE

Creates a copy of an aggregate and its entire substructure. If the specified aggregate is part of a sequence, only the aggregate specified and its substructure, rather than the entire sequence, is copied.

## VAX FORMAT

**status = cda$copy_aggregate**

*(root-aggregate-handle ,input-aggregate-handle ,output-aggregate-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **input-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **output-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

# COPY AGGREGATE

## C FORMAT

**status = CdaCopyAggregate**

*(root_aggregate_handle, input_aggregate_handle, output_aggregate_handle)*

## Argument Information

```
unsigned long CdaCopyAggregate(root_aggregate_handle,
                      input_aggregate_handle,
                      output_aggregate_handle)
              unsigned long       root_aggregate_handle;
              unsigned long       input_aggregate_handle;
              unsigned long       *output_aggregate_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate with which the copied aggregate is associated. 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***
Identifier of the aggregate to be copied.

***output-aggregate-handle***
Receives the handle of the new copy of the specified aggregate. This new aggregate handle must be used in all subsequent operations on that aggregate.

## Description

The COPY AGGREGATE routine makes a copy of the specified aggregate and its entire substructure. 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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_INVAGGTYP | Invalid aggregate type |

# CREATE AGGREGATE

Creates a new aggregate that contains empty items. Once this aggregate is created, it can be populated using the STORE ITEM routine.

## VAX FORMAT

**status = cda$create_aggregate**

*(root-aggregate-handle ,aggregate-type ,aggregate-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-type** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaCreateAggregate**

*(root_aggregate_handle, aggregate_type, aggregate_handle)*

## Argument Information

```
unsigned long CdaCreateAggregate(root_aggregate_handle,
                  aggregate_type, aggregate_handle)
              unsigned long      root_aggregate_handle;
              unsigned long      aggregate_type;
              unsigned long      *aggregate_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate with which the newly created aggregate is associated. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

***aggregate-type***
The type of aggregate to be created, expressed as a symbolic constant. The aggregate type symbolic constants are defined in the files ddif$def.h and dtif$def.h on VMS systems and in the files ddif_def.h and dtif_def.h on ULTRIX systems.

***aggregate-handle***
Receives the identifier of the newly created aggregate. This handle must be used in all subsequent operations on that aggregate.

## Description

The CREATE AGGREGATE routine creates a new aggregate of the specified type 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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_INVAGGTYP | Invalid aggregate type |

Any error returned by the memory allocation routines.

# CREATE FILE

Creates a new compound document file for output. An output stream is also created.

## VAX FORMAT

**status = 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)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **file-spec-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **file-spec** | VMS usage: | char_string |
| | Data type: | character string |
| | Access: | read only |
| | Mechanism: | by reference |
| **default-file-spec-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

| Argument | Argument Information | |
|---|---|---|
| **default-file-spec** | VMS usage: | char_string |
| | Data type: | character string |
| | Access: | read only |
| | Mechanism: | by reference |
| **alloc-rtn** | VMS usage: | procedure |
| | Data type: | procedure entry mask |
| | Access: | call after stack unwind |
| | Mechanism: | by reference, procedure reference |
| **dealloc-rtn** | VMS usage: | procedure |
| | Data type: | procedure entry mask |
| | Access: | call after stack unwind |
| | Mechanism: | by reference, procedure reference |
| **alloc-dealloc-prm** | VMS usage: | context |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **result-file-spec-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **result-file-spec** | VMS usage: | char_string |
| | Data type: | character string |
| | Access: | write only |
| | Mechanism: | by reference |
| **result-file-ret-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

| Argument | Argument Information | |
|---|---|---|
| **stream-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **file-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

### status = CdaCreateFile

*(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)*

## Argument Information

```
unsigned long CdaCreateFile(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)
                unsigned long       file_spec_len;
                unsigned char       *file_spec;
                unsigned long       default_file_spec_len;
                unsigned char       *default_file_spec;
                unsigned long       (*alloc_rtn)();
                unsigned long       (*dealloc_rtn)();
                unsigned long       alloc_dealloc_prm;
                unsigned long       root_aggregate_handle;
                unsigned long       result_file_spec_len;
                unsigned char       *result_file_spec;
                unsigned long       *result_file_ret_len;
                unsigned long       *stream_handle;
                unsigned long       *file_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***file-spec-len***
The length (in bytes) of the string specified by the **file-spec** parameter.

***file-spec***
The file specification.

***default-file-spec-len***
The length (in bytes) of the buffer specified by **default-file-spec**. 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***
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. On ULTRIX systems, the string is appended to the file specification, if the file specification does not already contain a period.

***alloc-rtn***
Address of a memory allocation routine. The calling sequence for an allocation routine is defined in the Description section of this routine. If you specify 0 for this argument, a default memory allocation routine is used. For a description, see Chapter 9.

***dealloc-rtn***
Address of a memory deallocation routine. The calling sequence for a deallocation routine is defined in the Description section of this routine. If you specify 0 for this argument, a default memory deallocation routine is used. For a description, see Chapter 9.

***alloc-dealloc-prm***
User context to be passed to the memory allocation and deallocation routines. If the system default memory allocation or deallocation routine is used, this parameter is ignored. For a description, see Chapter 9.

***root-aggregate-handle***
Identifier of the root aggregate associated with the newly created compound document. This handle must be used in all subsequent operations on that root aggregate. This handle is returned by a call to the CREATE ROOT AGGREGATE routine.

The **root-aggregate-handle** argument is used to specify the file type of the newly created document using the aggregate type DDIF$_DDF for a DDIF file or DTIF$_DTF for a DTIF file.

***result-file-spec-len***
Length (in bytes) of the buffer specified by **result-file-spec**. If you specify 0 for this parameter, the length of the resultant file specification is not returned.

***result-file-spec***
Receives the resultant 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. On ULTRIX systems, the **file-spec** argument is copied to this buffer.

***result-file-ret-len***
Receives the actual length (in bytes) 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***
Receives the handle of the newly created compound document stream. This handle must be used in all subsequent operations on that stream.

***file-handle***
Receives the handle of the newly created compound document file. 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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| 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

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 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 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.

```
    .
    .
    .
/* 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);
    .
    .
    .
```

# CREATE ROOT AGGREGATE

Creates a document root aggregate.

## VAX FORMAT

**status = cda$create_root_aggregate**

*(alloc-rtn ,dealloc-rtn ,alloc-dealloc-prm ,processing-options ,aggregate-type ,root-aggregate-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **alloc-rtn** | VMS usage: | procedure |
| | Data type: | procedure entry mask |
| | Access: | call after stack unwind |
| | Mechanism: | by reference, procedure reference |
| **dealloc-rtn** | VMS usage: | procedure |
| | Data type: | procedure entry mask |
| | Access: | call after stack unwind |
| | Mechanism: | by reference, procedure reference |
| **alloc-dealloc-prm** | VMS usage: | context |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by value |

| Argument | Argument Information | |
|---|---|---|
| **processing-options** | VMS usage: | item_list_2 |
| | Data type: | record |
| | Access: | read only |
| | Mechanism: | by reference, array reference |
| **aggregate-type** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaCreateRootAggregate**

*(alloc_rtn, dealloc_rtn, alloc_dealloc_prm, processing_options, aggregate_type, root_aggregate_handle)*

## Argument Information

```
unsigned long CdaCreateRootAggregate(alloc_rtn, dealloc_rtn,
                    alloc_dealloc_prm, processing_options,
                    aggregate_type, root_aggregate_handle)
               unsigned long     (*alloc_rtn)();
               unsigned long     (*dealloc_rtn)();
               unsigned long     *processing_options;
               unsigned long     alloc_dealloc_prm;
               unsigned long     aggregate_type;
               unsigned long     *root_aggregate_handle;
```

## RETURNS

***status***

A condition value indicating the return status of the routine call.

## Arguments

***alloc-rtn***
Address of a memory allocation routine. The calling sequence for an allocation routine is defined in the Description section of this routine. If you specify 0 for this argument, a default memory allocation routine is used. For a description, see Chapter 9.

***dealloc-rtn***
Address of a memory deallocation routine. The calling sequence for a deallocation routine is defined in the Description section of this routine. If you specify 0 for this argument, a default memory deallocation routine is used. For a description, see Chapter 9.

***alloc-dealloc-prm***
User context to be passed to the memory allocation and deallocation routines. If the system default memory allocation or deallocation routine is used, this parameter is ignored. For a description, see Chapter 9.

***processing-options***
An item list containing options to control input processing. 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$_INHERIT_ATTRIBUTES | Attribute inheritance is applied to all document segments. First, if a segment has a type reference that corresponds to a type definition, the attributes of the type are applied to the segment.<br><br>If a segment is the root segment, and a style guide is referenced in the document's header, the definitions and layout from the style guide are applied to the root segment. For the root segment only, standard defined initial values are applied to the attributes of the segment that do not yet have values.<br><br>If the segment is not the root segment, attribute values of its parent segment are applied to the attributes of the segment that do not yet have values. For more information on the inherit attributes processing option, see Section 1.6.1. |
| DDIF$_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. For more information on the retain definitions processing option, see Section 1.6.2. |

| | |
|---|---|
| DDIF$_EVALUATE_CONTENT | Content reference (DDIF$_CRF) aggregates are replaced with the value of the definition (DDIF$_CTD) they reference. The value of this content definition may be in the current document or in an external document.<br>Content for segments with the DDIF$_SGA_COMPUTE_C item present in the segment's attributes (DDIF$_SGA) may be imported from an external reference. If the value of the DDIF$_SGA_COMPUTE_C item is DDIF$K_REMOTE_COMPUTE, the external content is imported and replaces the segment's original content. If the value of the DDIF$_SGA_COMPUTE_C item is DDIF$_K_COPY_COMPUTE, the external content is imported only if the segment has no content. For more information on the evaluate content processing option, see Section 1.6.3. |
| DDIF$_DISCARD_I_SEGMENTS | Segments of the image ( $I ) content category, and any nested segments, are discarded. For more information on the discard image segments processing option, see Section 1.6.4. |
| DDIF$_DISCARD_2D_SEGMENTS | Segments of the graphics ( $2D ) content category, and any nested segments, are discarded. For more information on the discard graphics segments processing option, see Section 1.6.4. |
| DDIF$_DISCARD_T_SEGMENTS | Segments of the text ( $T ) content category, and any nested segments, are discarded. For more information on the discard text segments processing option, see Section 1.6.4. |
| DDIF$_DISCARD_TBL_SEGMENTS | Segments of the table ( $TBL ) content category, and any nested segments, are discarded. For more information on the discard table segments processing option, see Section 1.6.4. |
| DDIF$_DISCARD_PDL_SEGMENTS | Segments of the page description language ( $PDL ) content category, and any nested segments, are discarded. For more information on the discard page descriptions language segments processing option, see Section 1.6.4. |

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***
The type of aggregate to be created, expressed as a symbolic constant. The only valid root aggregate types are DDIF$_DDF and DTIF$_DTF.

***root-aggregate-handle***
Receives a value that identifies the newly created root aggregate. This handle must be used in all subsequent operations on that root aggregate.

## Description

The CREATE ROOT AGGREGATE routine creates a document root aggregate.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_INVAGGTYP | Invalid aggregate type |
| CDA$_INVITMLST | Invalid item list |

Any error returned by the memory allocation routines.

## Example

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 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.

```
    .
    .
    .
aggregate_type = DDIF$_DDF;
status = cda$create_root_aggregate(0, 0, 0, 0, &aggregate_type,
                                   &root_aggregate_handle);
if (FAILURE(status)) return(status);
    .
    .
    .
```

# CREATE STREAM

Opens a compound document stream for output.

## VAX FORMAT

**status = cda$create_stream**

*(alloc-rtn ,dealloc-rtn ,alloc-dealloc-prm ,put-rtn ,put-prm ,buf-len ,buf-adr ,stream-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **alloc-rtn** | VMS usage: | procedure |
| | Data type: | procedure entry mask |
| | Access: | call after stack unwind |
| | Mechanism: | by reference, procedure reference |
| **dealloc-rtn** | VMS usage: | procedure |
| | Data type: | procedure entry mask |
| | Access: | call after stack unwind |
| | Mechanism: | by reference, procedure reference |
| **alloc-dealloc-prm** | VMS usage: | context |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by value |

| Argument | Argument Information | |
|---|---|---|
| **put-rtn** | VMS usage: | procedure |
| | Data type: | procedure entry mask |
| | Access: | read only |
| | Mechanism: | by reference, procedure reference |
| **put-prm** | VMS usage: | user_arg |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by value |
| **buf-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **buf-adr** | VMS usage: | vector_byte_unsigned |
| | Data type: | byte (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference, array reference |
| **stream-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaCreateStream**

*(alloc_rtn, dealloc_rtn, alloc_dealloc_prm, put_rtn, put_prm, buf_len, buf_adr, stream_handle)*

# CREATE STREAM

## Argument Information

```
unsigned long CdaCreateStream(alloc_rtn, dealloc_rtn,
                    alloc_dealloc_prm, put_rtn, put_prm,
                    buf_len, buf_adr, stream_handle)
          unsigned long       (*alloc_rtn)();
          unsigned long       (*dealloc_rtn)();
          unsigned long       alloc_dealloc_prm;
          unsigned long       (*put_rtn)();
          unsigned long       put_prm;
          unsigned long       buf_len;
          unsigned char       *buf_adr;
          unsigned long       *stream_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***alloc-rtn***
Address of a memory allocation routine. The calling sequence for an allocation routine is defined in the Description section of this routine. If you specify 0 for this argument, a default memory allocation routine is used. For a description, see Chapter 9.

***dealloc-rtn***
Address of a memory deallocation routine. The calling sequence for a deallocation routine is defined in the Description section of this routine. If you specify 0 for this argument, a default memory deallocation routine is used. For a description, see Chapter 9.

***alloc-dealloc-prm***
User context to be passed to the memory allocation and deallocation routines. If the system default memory allocation or deallocation routine is used, this parameter is ignored. For a description, see Chapter 9.

***put-rtn***
Address of a stream *put* routine. The calling sequence for a *put* routine is defined in Chapter 9. If you specify 0 for this argument on VMS systems, a system default routine is used. On ULTRIX systems, you must provide both a put-rtn and put-prm; there is no default. 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***
User context to be passed to the stream *put* routine. If the VMS system default *put* routine is used, the value must be a pointer to a RAB. On ULTRIX systems, you must provide both a put-rtn and put-prm; there is no default. For a description, see Chapter 9.

***buf-len***
Length of the buffer specified by the **buf-adr** parameter.

***buf-adr***
Address of a buffer that receives the output data.

***stream-handle***
Receives the handle of the newly created stream. 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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

Any error returned by the memory allocation routines.

# CREATE TEXT FILE

Creates a standard text file for output.

## VAX FORMAT

**status = 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)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **file-spec-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **file-spec** | VMS usage: | char_string |
| | Data type: | character string |
| | Access: | read only |
| | Mechanism: | by reference |
| **default-file-spec-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **default-file-spec** | VMS usage: | char_string |
| | Data type: | character string |
| | Access: | read only |
| | Mechanism: | by reference |

| Argument | Argument Information | |
|---|---|---|
| **result-file-spec-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **result-file-spec** | VMS usage: | char_string |
| | Data type: | character string |
| | Access: | write only |
| | Mechanism: | by reference |
| **result-file-ret-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **text-file-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaCreateTextFile**

*(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)*

# CREATE TEXT FILE

## Argument Information

```
unsigned long CdaCreateTextFile(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)
            unsigned long        file_spec_len;
            unsigned char        *file_spec;
            unsigned long        default_file_spec_len;
            unsigned char        *default_file_spec;
            unsigned long        result_file_spec_len;
            unsigned char        *result_file_spec;
            unsigned long        *result_file_ret_len;
            unsigned long        *text_file_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***file-spec-len***
Length (in bytes) of the string specified by the **file-spec** argument.

***file-spec***
File specification of the text file to be created for output.

***default-file-spec-len***
Length (in bytes) of the string specified by **default-file-spec**. If you specify 0 for this parameter, no default file specification is used.

***default-file-spec***
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. On ULTRIX systems, the string is appended to the file specification if the file specification does not already contain a period.

***result-file-spec-len***
Length (in bytes) of the buffer specified by **result-file-spec**. If you specify 0 for this parameter, the length of the resultant file specification is not returned.

***result-file-spec***
Receives the resultant file specification. This file specification is the result of a VMS RMS $CREATE operation. On ULTRIX systems, the **file-spec** is copied to this buffer. If you specify 0 for this parameter, a resultant file specification is not returned.

***result-file-ret-len***
Receives the actual length (in bytes) 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***
Receives the handle of the text file. 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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

Any error returned by the memory allocation routines.

Any error returned by the file routines.

# 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.

## VAX FORMAT

**status = cda$delete_aggregate**
*(root-aggregate-handle ,aggregate-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaDeleteAggregate**
*(root_aggregate_handle, aggregate_handle)*

## Argument Information

```
unsigned long CdaDeleteAggregate(root_aggregate_handle,
                  aggregate_handle)
            unsigned long      root_aggregate_handle;
            unsigned long      aggregate_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate associated with the aggregate to be deleted. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

***aggregate-handle***
Identifier of the aggregate to be destroyed.

## 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 to children of the root aggregate are invalid after a call to this routine.

## RETURN VALUES

| **Return Value** | **Description** |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_INVAGGTYP | Invalid aggregate type |

Any error returned by the memory deallocation routines.

# DELETE ROOT AGGREGATE

Destroys a document root aggregate and all of its substructure.

## VAX FORMAT

**status = cda$delete_root_aggregate**
*(root-aggregate-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaDeleteRootAggregate**
*(root_aggregate_handle)*

## Argument Information

```
unsigned long CdaDeleteRootAggregate(root_aggregate_handle)
              unsigned long       root_aggregate_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate to be deleted. 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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

Any error returned by the memory deallocation routines.

# ENTER SCOPE

Opens a document scope for incremental writing.

## VAX FORMAT

**status = cda$enter_scope**

*(root-aggregate-handle ,stream-handle ,scope-code [,aggregate-handle])*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **stream-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **scope-code** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaEnterScope**

*(root_aggregate_handle, stream_handle, scope_code, aggregate_handle)*

## Argument Information

```
unsigned long CdaEnterScope(root_aggregate_handle,
          stream_handle, scope_code
          aggregate_handle)
     unsigned long       root_aggregate_handle;
     unsigned long       stream_handle;
     unsigned long       scope_code;
     unsigned long       aggregate_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate associated with the document content to be incrementally written. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

***stream-handle***
Identifier of the stream associated with the document to be written. This handle is returned by a call to either the CREATE FILE routine or the CREATE STREAM routine.

***scope-code***
Symbolic constant identifying the scope to be opened. Valid values for the DDIF root aggregate are as follows:

| Code | Meaning |
|---|---|
| DDIF$K_DOCUMENT_SCOPE | Document scope |
| DDIF$K_CONTENT_SCOPE | Content scope |
| DDIF$K_SEGMENT_SCOPE | Segment scope |

Valid values for the DTIF root aggregate are as follows:

| Code | Meaning |
|---|---|
| DTIF$K_DOCUMENT_SCOPE | Document scope |
| DTIF$K_TABLE_SCOPE | Table scope |
| DTIF$K_ROW_SCOPE | Row scope |
| DTIF$K_CELLS_SCOPE | Cell scope (for all cells in a row) |

***aggregate-handle***
Identifier of an aggregate of the appropriate type, if required by the scope code specified.

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. |

The DTIF scoped sections that require that the **aggregate-handle** be specified are as follows:

| Scope | Value of Aggregate-Handle |
|---|---|
| DTIF$K_TABLE_SCOPE | **Aggregate-handle** is the handle of the table aggregate, which contains everything to be specified except the rows. |
| DTIF$K_ROW_SCOPE | **Aggregate-handle** is the handle of the row aggregate, which contains everything to be specified except the cells. |

## 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 DDIF-encoded document are the following:

- DDIF$K_DOCUMENT_SCOPE
- DDIF$K_CONTENT_SCOPE
- DDIF$K_SEGMENT_SCOPE

For a DTIF-encoded document, the types of scopes that you can open are as follows:

- DTIF$K_DOCUMENT_SCOPE
- DTIF$K_TABLE_SCOPE
- DTIF$K_ROW_SCOPE
- DTIF$K_CELLS_SCOPE

### Using Scope to Write DDIF Documents Incrementally

When performing incremental writing on a DDIF-encoded document, 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 following methods 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 |

### Using Scope to Write DTIF Documents Incrementally

When performing incremental writing of a DTIF-encoded document, you should perform the following steps:

1. Call the ENTER SCOPE routine, specifying **scope-code** as DTIF$K_DOCUMENT_SCOPE.
2. Create a header (type DTIF$_HDR) aggregate and write it using the PUT AGGREGATE routine.
3. Create a table (DTIF$_TBL) aggregate, specifying everything to be written except the table rows.
4. Call the ENTER SCOPE routine, specifying **scope-code** as DTIF$K_TABLE_SCOPE and **aggregate-handle** as the handle of the table aggregate to be written.
5. Create a row (DTIF$_ROW) aggregate, specifying everything to be written except the row cells.
6. Call the ENTER SCOPE routine, specifying **scope-code** as DTIF$K_ROW_SCOPE and **aggregate-handle** as the handle of the row aggregate to be written.
7. Call the ENTER SCOPE routine, specifying **scope-code** as DTIF$K_CELLS_SCOPE (do not specify the **aggregate-handle** argument).
8. Create and populate a cell (DTIF$_CLD) aggregate, and invoke the PUT AGGREGATE routine to write the completed aggregate.
9. Repeat until all of the cells in the row have been written.
10. Call the LEAVE SCOPE routine, specifying **scope-code** as DTIF$K_CELLS_SCOPE.
11. Call the LEAVE SCOPE routine, specifying **scope-code** as DTIF$K_ROW_SCOPE.
12. Repeat steps 6 through 11 for each row in the table.
13. Once all the rows in the table have been completed, call the LEAVE SCOPE routine, specifying **scope-code** as DTIF$K_TABLE_SCOPE.

14. If there are additional tables to be created, repeat steps 4 through 13 to create the additional tables.
15. Once all the tables in the document have been created, call the LEAVE SCOPE routine, specifying **scope-code** as DTIF$K_DOCUMENT_SCOPE.

**NOTE**

After calling the ENTER SCOPE routine, if your application no longer requires the aggregates written, you should issue a subsequent call to the DELETE AGGREGATE routine to destroy these aggregates.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_INVSCOCOD | Invalid scope code |

Any errors returned by the file routines.

## Examples

The following example shows how to incrementally read a DDIF document and write it to another DDIF file using the PUT AGGREGATE, ENTER SCOPE, and LEAVE SCOPE routines.

```
                .
                .
                .
/* 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:
            /* The first aggregate is incrementally read--enter
               the document scope here before putting out the
               aggregate */
                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;
```

## Examples

```
/* 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$K_CONTENT_SCOPE;
     /* DDIF$_DHD immediately precedes content--enter
        content scope here */
        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$K_SEGMENT_SCOPE;
     /* Enter segment scope passing segment handle--
        this call outputs the segment aggregate--enter
        scope does put aggregate for segments */
         status = cda$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 */

           .
           .
           .
```

The following example shows the incremental method of creating a document, using both methods outlined for writing nested segments.

```
/*

   This is an example of using the incremental method to create a
   document with nested segments being output using different options.

*/
#ifdef vms
#include <cda$def.h>
#include <ddif$def.h>
#else
#include <cda_def.h>
#include <ddif_def.h>
#endif

#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 segment" 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 its 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:

```
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
   }
  }
 }
}
```

# 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.

## VAX FORMAT

**status = cda$erase_item**

*(root-aggregate-handle ,aggregate-handle ,aggregate-item [,aggregate-index])*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-item** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

| Argument | Argument Information | |
|---|---|---|
| **aggregate-index** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaEraseItem**

*(root_aggregate_handle, aggregate_handle, aggregate_item, aggregate_index)*

## Argument Information

```
unsigned long CdaEraseItem(root_aggregate_handle,
            aggregate_handle, aggregate_item,
            aggregate_index)
           unsigned long     root_aggregate_handle;
           unsigned long     aggregate_handle;
           unsigned long     aggregate_item;
           unsigned long     aggregate_index;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate of which the aggregate containing the item is a part. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

***aggregate-handle***
Identifier of the aggregate containing the item to be erased.

***aggregate-item***
Identifying code of the item to be erased, expressed as a symbolic constant. The DDIF aggregate item symbolic constants are defined in the file ddif$def.h on VMS systems and in the file ddif_def.h on ULTRIX systems and are discussed in Chapter 4. The DTIF aggregate item symbolic codes are defined in the file

# ERASE ITEM

dtif$def.h on VMS systems and in the file dtif_def.h on ULTRIX systems and are described in Chapter 5.

***aggregate-index***
Index of the item to be erased (relative to 0). 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 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 deleted.

## RETURN VALUES

| Return Value | Description |
|---|---|
| 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. |

# FIND DEFINITION

Looks up the specified definition in a list of definitions.

## VAX FORMAT

**status = cda$find_definition**

*(root-aggregate-handle ,aggregate-type ,buf-len ,buf-adr ,aggregate-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-type** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **buf-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **buf-adr** | VMS usage: | vector_byte_unsigned |
| | Data type: | byte (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference, array reference |

| Argument | Argument Information | |
|---|---|---|
| **aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaFindDefinition**

*(root_aggregate_handle, aggregate_type, buf_len, buf_adr, aggregate_handle)*

## Argument Information

```
unsigned long CdaFindDefinition(root_aggregate_handle,
                    aggregate_type, buf_len, buf_adr,
                    aggregate_handle)
              unsigned long        root_aggregate_handle;
              unsigned long        aggregate_type;
              unsigned long        buf_len;
              unsigned char        *buf_adr;
              unsigned long        *aggregate_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate with which the definition aggregate being searched for is associated. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

***aggregate-type***
The type of definition aggregate being searched for, expressed as a symbolic constant. The DDIF aggregate type symbolic constants are defined in the file ddif$def.h on VMS systems and in the file ddif_def.h on ULTRIX systems and are discussed in Chapter 4. The DTIF aggregate type symbolic codes are defined in the file dtif$def.h on VMS systems and in the file dtif_def.h on ULTRIX systems and are described in Chapter 5.

***buf-len***

Length of the buffer (in bytes) specified by **buf-adr**.

***buf-adr***

The buffer that contains the selector value used to indicate the desired definition from the list of definitions. 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***

Receives a value that identifies the newly located definition aggregate. 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.

# FIND DEFINITION

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_INVAGGTYP | Invalid aggregate type |
| CDA$_INVBUFLEN | Invalid buffer length |
| CDA$_DEFNOTFOU | Definition not found |

# FIND TRANSFORMATION

Returns the current transformation matrix values.

## VAX FORMAT

**status = cda$find_transformation**
*(root-aggregate-handle ,transformation)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **transformation** | VMS usage: | address |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaFindTransformation**
*(root_aggregate_handle, transformation)*

# FIND TRANSFORMATION

## Argument Information

```
unsigned long CdaFindTransformation(root_aggregate_handle,
                           transformation)
            unsigned long       root_aggregate_handle;
            float               **transformation;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

***transformation***
Receives the address of a vector of nine single-precision floating-point elements.

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 specified by the DDIF$_TRN 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

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_DEFNOTFOU | Definition not found |

# FLUSH STREAM

Flushes the contents of the stream and ensures that the data has been physically transferred to the receiving medium.

## VAX FORMAT

**status = cda$flush_stream**

*(stream-handle ,flush-rtn ,flush-prm)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **stream-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **flush-rtn** | VMS usage: | procedure |
| | Data type: | procedure entry mask |
| | Access: | call after stack unwind |
| | Mechanism: | by reference, procedure reference |
| **flush-prm** | VMS usage: | context |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by value |

## C FORMAT

**status = CdaFlushStream**

*(stream_handle, flush_rtn, flush_prm)*

## Argument Information

```
unsigned long CdaFlushStream(stream_handle,
                flush_rtn, flush_prm)
             unsigned long      stream_handle;
             unsigned long      (*flush_rtn)();
             unsigned long      flush_prm;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***stream-handle***
Identifier of the output stream to be flushed. This handle is returned by a call to the CREATE STREAM routine.

***flush-rtn***
Address of a stream *flush* routine. If you specify 0 for this argument, a default flush-rtn 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. For more information, see Chapter 9.

***flush-prm***
User context to be passed to the stream *flush* routine. This argument should contain the value of the **put-prm** argument passed in a call to the CREATE STREAM routine. For more information, see Chapter 9.

## 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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

Any error returned by the file routines.

# GET AGGREGATE

Reads the next aggregate from the specified stream.

## VAX FORMAT

**status = cda$get_aggregate**

*(root-aggregate-handle ,stream-handle ,aggregate-handle ,aggregate-type)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **stream_handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **aggregate-type** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaGetAggregate**

*(root_aggregate_handle, stream_handle, aggregate_handle, aggregate_type)*

## Argument Information

```
unsigned long CdaGetAggregate(root_aggregate_handle,
            stream_handle, aggregate_handle,
            aggregate_type)
        unsigned long       root_aggregate_handle;
        unsigned long       stream_handle;
        unsigned long       *aggregate_handle;
        unsigned long       *aggregate_type;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate associated with the aggregate to be read. 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 the aggregates, you cannot specify the same **root-aggregate-handle** again when calling this routine.

***stream-handle***
Identifier of the stream from which the aggregate is to be read. This handle is returned by a call to either the OPEN FILE routine or the OPEN STREAM routine.

***aggregate-handle***
Receives the handle of the retrieved aggregate. This aggregate handle is used to identify the retrieved aggregate to any other aggregate transfer procedure.

***aggregate-type***
Receives the aggregate type. The DDIF aggregate type symbolic codes are defined in the file ddif$def.h on VMS systems and in the file ddif_def.h on ULTRIX systems and are described in Chapter 4. The DTIF aggregate type symbolic codes

are defined in the file dtif$def.h on VMS systems and in the file dtif_def.h on ULTRIX systems and are described in Chapter 5.

Valid aggregate types are any one of the primary DDIF or DTIF aggregates:

| 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$_CRF | Content reference |
| DDIF$_PVT | Private content |
| DDIF$_GLY | Layout galley |
| DDIF$_EOS | End of segment |
| DDIF$_EXT | External content |
| DTIF$_DSC | Document descriptor |
| DTIF$_HDR | Document header |
| DTIF$_TBL | Table definition |
| DTIF$_ROW | Row definition |
| DTIF$_CLD | Cell data |

These aggregates are the only aggregates that can be returned by the GET AGGREGATE routine. All other aggregates are somehow connected to these aggregates and can be located by traversing the structure using other routines (such as LOCATE ITEM and NEXT AGGREGATE). If the aggregate type is DDIF$_EOS (end of segment), the **aggregate-handle** is 0 to indicate that the nested segment has been completed.

## Description

The GET AGGREGATE routine reads the next primary aggregate from a specified stream. (The primary aggregates are listed in the description of the **aggregate-type** argument.)

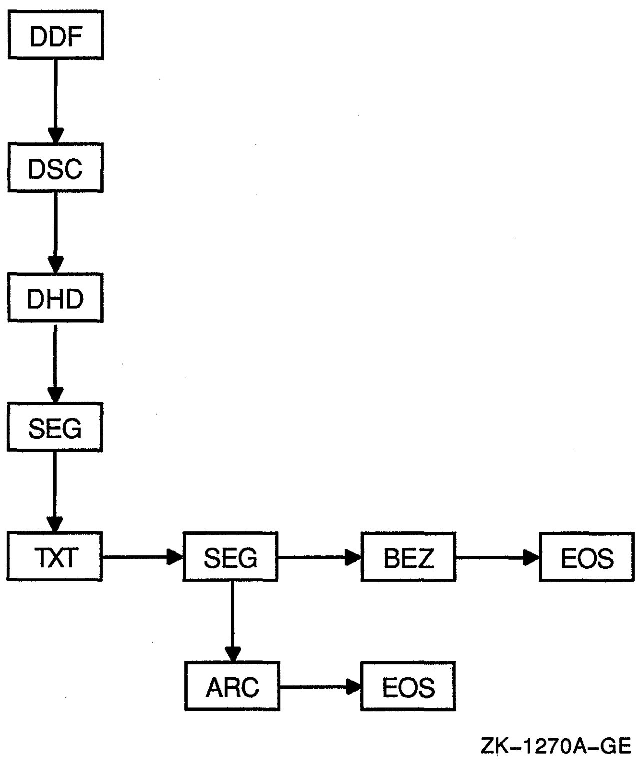
The GET AGGREGATE routine has three restrictions on the information returned:

- The GET AGGREGATE routine returns only primary aggregates (as listed in the description of the **aggregate-type** argument). Other aggregates are returned attached to the primary aggregates.
- When you call the GET AGGREGATE routine for a segment aggregate, it returns all the items in the segment aggregate (and all its substructure) **except for the content item (DDIF$_SEG_CONTENT) and its substructure.**
- When you call the GET AGGREGATE routine and it returns a DDIF$_EXT aggregate, the encoding items (DDIF$_EXT_ENCODING_C, DDIF$_EXT_ENCODING, and DDIF$_EXT_ENCODING_L will be empty. A call to the GET EXTERNAL ENCODING routine must be made before the next call to get AGGREGATE.
- Content aggregates are returned one at a time, following the segment aggregate.
- When you are processing a document, you must observe the occurrence of DDIF$_EOS aggregates, which denote the end of a segment's content. It is also important to note that the DDIF$_EOS aggregate is a **dummy** aggregate; it is not an actual aggregate and therefore does not have a valid aggregate handle. Instead, it is simply an aggregate type that is returned to indicate the end of a segment. The next aggregate returned is a sibling to that segment.

The GET AGGREGATE routine reads the primary aggregates in a document in a hierarchical fashion. That is, whenever GET AGGREGATE encounters a segment, your next call to GET AGGREGATE descends to the next level of the 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.

For example, consider a document that 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). This document is illustrated in Figure 8–1.

**Figure 8–1: Example Document**




Following these generalized rules, 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)

## RETURN VALUES

| Return Value | Description |
|---|---|
| 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.

# GET ARRAY SIZE

Determines the number of elements present in an array-valued aggregate item.

## VAX FORMAT

**status = cda$get_array_size**

*(aggregate-handle ,aggregate-item ,array-size)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-item** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **array-size** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaGetArraySize**

*(aggregate_handle, aggregate_item, array_size)*

## Argument Information

```
unsigned long CdaGetArraySize(aggregate_handle,
                    aggregate_item, array_size)
              unsigned long       aggregate_handle;
              unsigned long       aggregate_item;
              unsigned long       *array_size;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***aggregate-handle***
Identifier of the aggregate containing the array-valued item.

***aggregate-item***
Identifying code of the array-valued aggregate item, expressed as a symbolic constant. The DDIF aggregate item symbolic constants are defined in the module ddif$def.h on VMS systems and in the module ddif_def.h on ULTRIX systems and are defined in Chapter 4. The DTIF aggregate type symbolic codes are defined in the file dtif$def.h on VMS systems and in the file dtif_def.h on ULTRIX systems and are described in Chapter 5.

***array-size***
Receives the number of elements present in the array-valued item. Because the index is zero based, this number is equal to 1 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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion. |
| CDA$_INVAGGTYP | Invalid aggregate type. |
| CDA$_INVITMCOD | Invalid item code. |
| CDA$_EMPTY | Item is empty. |

# GET DOCUMENT

Reads an entire compound document from the specified stream.

## VAX FORMAT

**status = cda$get_document**

*(root-aggregate-handle ,stream-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **stream-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaGetDocument**

*(root_aggregate_handle, stream_handle)*

# GET DOCUMENT

## Argument Information

```
unsigned long CdaGetDocument(root_aggregate_handle,
                    stream_handle)
               unsigned long       root_aggregate_handle;
               unsigned long       stream_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate associated with the document to be read. 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***
Identifier of the stream from which the document is to be read. 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 memory in aggregates that are linked from the document root aggregate.

## RETURN VALUES

| Return Value | Description |
|---|---|
| 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.

# GET EXTERNAL ENCODING

Reads the value of an external encoding from the specified stream and stores it as the value of the **agg$**_EXT_ENCODING item in the appropriate aggregate, which can be DDIF$_EXT, DTIF$_EXT, or ESF$_EXT.

## VAX FORMAT

**status = cda$get_external_encoding**

*(root-aggregate-handle ,stream-handle ,aggregate-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **stream-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | modify |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaGetExternalEncoding**

*(root_aggregate_handle, stream_handle, aggregate_handle)*

## Argument Information

```
unsigned long CdaGetExternalEncoding(root_aggregate_handle,
                    stream_handle, aggregate_handle)
              unsigned long       root_aggregate_handle;
              unsigned long       stream_handle;
              unsigned long       *aggregate_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

***stream-handle***
Identifier of the stream containing the external encoding. This handle is returned by a call to either the OPEN FILE routine or the OPEN STREAM routine.

***aggregate-handle***
Identifier of an aggregate of type DDIF$_EXT, DTIF$_EXT, or ESF$_EXT. The external encoding value that is read from the stream is written to the **agg**$_EXT_ENCODING item in the appropriate aggregate, where **agg** refers to the specific aggregate type. That 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 **agg**$_EXT_ENCODING item of the aggregate specified by **aggregate-handle**, which can be an aggregate of type DDIF$_EXT, DTIF$_EXT, or ESF$_EXT. If the external encoding is DDIF or DTIF, the value stored is the handle of a DDIF or DTIF root aggregate, which contains the entire document in the external encoding.

# GET EXTERNAL ENCODING

If used, the GET EXTERNAL ENCODING routine must be invoked immediately after the specified aggregate 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 on an inner document root aggregate.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_INVDOC | Invalid document |
| CDA$_INVAGGTYP | Invalid aggregate type |

# GET STREAM POSITION

Returns the current position in and size of a CDA data stream.

## VAX FORMAT

**status = cda$get_stream_position**

*(stream-handle ,position-rtn ,position-prm ,stream-position ,stream-size)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **stream-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **position-rtn** | VMS usage: | procedure |
| | Data type: | procedure entry mask |
| | Access: | call after stack unwind |
| | Mechanism: | by reference |
| **position-prm** | VMS usage: | context |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by value |
| **stream-position** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

# GET STREAM POSITION

| Argument | Argument Information | |
|---|---|---|
| **stream-size** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaGetStreamPosition**
*(stream_handle, position_rtn, position_prm, stream_position, stream_size)*

## Argument Information

```
unsigned long CdaGetStreamPosition(stream_handle,
            position_rtn, position_prm,
            stream_position, stream_size)
        unsigned long        stream_handle;
        unsigned long        (*position_rtn)();
        unsigned long        position_prm;
        unsigned long        *stream_position;
        unsigned long        *stream_size;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***stream-handle***
Identifier of the stream. The handle is returned by a call to either the OPEN STREAM routine or the OPEN FILE routine.

***position-rtn***
Address of a *get-position* routine. The calling sequence for a *get-position* routine is defined in Chapter 11. 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***
User context to be passed to the *get-position* routine. 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***
Receives the current position (in bytes) as measured from the start of the input stream being processed.

***stream-size***
Receives the total size (in bytes) of the input stream being processed.

## Description

The GET STREAM POSITION routine returns the current position and total size of the CDA data stream being processed.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

# GET TEXT POSITION

Returns the current position in and size of a text file.

## VAX FORMAT

**status = cda$get_text_position**

*(file-handle ,file-position ,file-size)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **file-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **file-position** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **file-size** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaGetTextPosition**

*(file_handle, file_position, file_size)*

## Argument Information

```
unsigned long CdaGetTextPosition(file_handle,
                file_position, file_size)
              unsigned long       file_handle;
              unsigned long       *file_position;
              unsigned long       *file_size;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***file-handle***
Identifier of the text file being processed. This handle is returned by a call to the OPEN TEXT FILE routine.

***file-position***
Receives the current position (in bytes) as measured from the start of the input text file being processed.

***file-size***
Receives the total size (in bytes) of the text file being processed.

## Description

The GET TEXT POSITION routine returns the current position in and total size of an input text file being processed.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

# 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.

## VAX FORMAT

**status = cda$insert_aggregate**

*(aggregate-handle ,prev-aggregate-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **prev-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaInsertAggregate**

*(aggregate_handle, prev_aggregate_handle)*

## Argument Information

```
unsigned long CdaInsertAggregate(aggregate_handle,
                                 prev_aggregate_handle)
                unsigned long      aggregate_handle;
                unsigned long      prev_aggregate_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***aggregate-handle***
Identifier of the aggregate to be inserted into the sequence.

***prev-aggregate-handle***
Identifier of the aggregate after which the aggregate identified by **aggregate-handle** is to be inserted in the sequence.

## 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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_INVINSERT | Aggregate already in a sequence |

## Example

This example illustrates the use of the INSERT AGGREGATE routine to insert an aggregate into a sequence.

```
    .
    .
    .
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);

    .
    .
    .
```

# LEAVE SCOPE

Completes a document that was incrementally written.

## VAX FORMAT

**status = cda$leave_scope**

*(root-aggregate-handle ,stream-handle ,scope-code)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **stream-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **scope-code** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaLeaveScope**

*(root_aggregate_handle, stream_handle, scope_code)*

## Argument Information

```
unsigned long CdaLeaveScope(root_aggregate_handle,
          stream_handle, scope_code)
     unsigned long       root_aggregate_handle;
     unsigned long       stream_handle;
     unsigned long       scope_code;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate associated with the document being incrementally written. This handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

***stream-handle***
Identifier of the stream associated with the document being incrementally written. This handle is returned by a call to either the CREATE FILE routine or the CREATE STREAM routine.

***scope-code***
Symbolic constant identifying the scope to be completed. Valid values are as follows:

| Code | Meaning |
|---|---|
| DDIF$K_DOCUMENT_SCOPE | Document scope |
| DDIF$K_CONTENT_SCOPE | Content scope |
| DDIF$K_SEGMENT_SCOPE | Segment scope |
| DTIF$K_DOCUMENT_SCOPE | Document scope |
| DTIF$K_TABLE_SCOPE | Table scope |
| DTIF$K_ROW_SCOPE | Row scope |
| DTIF$K_CELLS_SCOPE | Cell scope (for all cells in a row) |

## 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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_INVSCOCOD | Invalid scope code |

Any errors returned by the file routines.

# LOCATE ITEM

Locates an item within an aggregate by returning its address.

## VAX FORMAT

**status = cda$locate_item**

*(root-aggregate-handle ,aggregate-handle ,aggregate-item ,item-address ,item-length [,aggregate-index] [,add-info])*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-item** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **item-address** | VMS usage: | address |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

# LOCATE ITEM

| Argument | Argument Information | |
|---|---|---|
| **item-length** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **aggregate-index** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **add-info** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read or write |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaLocateItem**

*(root_aggregate_handle, aggregate_handle, aggregate_item, item_address, item_length, aggregate_index, add_info)*

## Argument Information

```
unsigned long CdaLocateItem(root_aggregate_handle,
                    aggregate_handle, aggregate_item,
                    item_address, item_length,
                    aggregate_index, add_info)
               unsigned long      root_aggregate_handle;
               unsigned long      aggregate_handle;
               unsigned long      aggregate_item;
               unsigned char      *item_address;
               unsigned long      *item_length;
               unsigned long      aggregate_index;
               unsigned long      *add_info;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate with which the aggregate containing the item to be located is associated. 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***
Identifier of the aggregate containing the item to be located.

***aggregate-item***
Identifying code of the item, expressed as a symbolic constant. The DDIF aggregate item symbolic constants are defined in the file ddif$def.h on VMS systems and in the file ddif_def.h on ULTRIX systems and are described in Chapter 4. The DTIF aggregate type symbolic codes are defined in the file dtif$def.h on VMS systems and in the file dtif_def.h on ULTRIX systems and are described in Chapter 5.

A user context item named DDIF$_USER_CONTEXT for DDIF aggregates and DTIF$_USER_CONTEXT for DTIF aggregates is available within every aggregate. This item is a longword that can be used by the application for any purpose.

For use by applications, a DDIF$_AGGREGATE_TYPE item and a DTIF$_ AGGREGATE_TYPE item are defined for every DDIF and DTIF aggregate type, respectively. It is a read-only item and, consequently, may be located using only this routine. If you specify this aggregate item, it returns the type of the aggregate.

***item-address***
Receives the address of the item's value. 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.

If the item being located contains an aggregate handle, a call to this routine returns the address of the aggregate handle. In order to use this aggregate handle, you must "dereference" it. For example, in C you would do the following:

```
cda$locate_item(&root_agg_handle, &agg_handle, &agg_item,
                &sub_agg, &item_length, &agg_index, &add_info);
```

The **sub_agg** parameter receives the address of the aggregate handle of the subaggregate. To use this handle, you would do the following:

```
agg_handle = *sub_agg;
cda$locate_item(&root_agg_handle, &agg_handle, &agg_item,
                &buf_addr, &buf_length, &agg_index, &add_info);
```

If there are subsequent aggregates in a sequence, you should use the NEXT AGGREGATE routine to retrieve the subsequent aggregates.

***item-length***
Receives the length (in bytes) of the item's value.

***aggregate-index***
Index of the item (relative to 0). 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***
Receives a data type-specific modifier for the data types character string and string with **add-info**. Selects the floating-point format to be returned for items with the data type general floating-point. Receives an integer scaling factor for the data type scaled integer. For data types other than character string, string with **add-info**, general floating-point, and scaled integer, 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. For the data type scaled integer, the **add-info** parameter receives an integer scaling factor. For the data type general floating-point, the **add-info** parameter contains a value that selects the format for the floating-point value to be returned. For **add-info** values for the general floating-point type, see Table 1–1. 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 the handle of an aggregate, you receive the address of the aggregate handle. To use this handle in subsequent routine calls, you must first "dereference" it. (For more information, see the description of the **item-address** argument.)

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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion. |
| CDA$_INVAGGTYP | Invalid aggregate type. |

| Return Value | Description |
|---|---|
| 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. |

# NEXT AGGREGATE

Locates the next aggregate in an aggregate sequence.

## VAX FORMAT

**status = cda$next_aggregate**

*(aggregate-handle ,next-aggregate-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **next-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaNextAggregate**

*(aggregate_handle, next_aggregate_handle)*

## Argument Information

```
unsigned long CdaNextAggregate(aggregate_handle,
                               next_aggregate_handle)
              unsigned long     aggregate_handle;
              unsigned long     *next_aggregate_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***aggregate-handle***
Identifier of the aggregate to be used in locating the next aggregate.

***next-aggregate-handle***
Receives the handle of the aggregate that follows the aggregate specified by **aggregate-handle**. 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 using the preceding aggregate as a reference. (The preceding aggregate is specified by the **aggregate-handle** argument.)

To read the aggregates in a sequence, you must retrieve the aggregate handle of the first aggregate using the LOCATE ITEM routine. (The handle of the first aggregate in the sequence is stored as an item in the current aggregate.) Once you have located the first item in the sequence using the LOCATE ITEM routine, 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$_TRN aggregates. To access the sequence of DDIF$_TRN aggregates, you would first use the LOCATE ITEM routine to read the handle of the first DDIF$_TRN aggregate that is stored in the DDIF$_CRF_TRANSFORM item. You would then use the NEXT AGGREGATE routine to return each additional aggregate in this encoded sequence, until the status CDA$_ENDOFSEQ is returned.

# NEXT AGGREGATE

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.

**NOTE**

If several different aggregate types may be linked in sequence, locate the aggregate type for the aggregate to determine its type (DDIF$_AGGREGATE_TYPE item code).

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_ENDOFSEQ | No successor aggregate found |

# OBJECT ID TO AGGREGATE TYPE

Translates an object identifier to a root aggregate type.

## VAX FORMAT

**status = cda$object_id_to_aggregate_type**

*(buf-len ,buf-adr ,nam-len ,nam-adr ,act-nam-len ,aggregate-type)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **buf-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **buf-adr** | VMS usage: | array |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference, array reference |
| **nam-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **nam-adr** | VMS usage: | array |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference, array reference |

# OBJECT ID TO AGGREGATE TYPE

| Argument | Argument Information | |
|---|---|---|
| **act-nam-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **aggregate-type** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaObjectIdToAggregateType**

*(buf_len, buf_adr, nam_len, nam_adr, act_nam_len, aggregate_type)*

## Argument Information

```
unsigned long CdaObjectIdToAggregateType(buf_len,
                   buf_adr, nam_len, nam_adr,
                   act_nam_len, aggregate_type)
              unsigned long       buf_len;
              unsigned long       buf_adr[];
              unsigned long       nam_len;
              unsigned char       *nam_adr;
              unsigned long       *act_nam_len;
              unsigned long       *aggregate_type;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***buf-len***
Length (in bytes) of the object identifier buffer.

***buf-adr***
Address of the object identifier.

***nam-len***
Length (in bytes) of the domain name buffer.

***nam-adr***
Receives the address of the domain name buffer.

***act-nam-len***
Receives the actual length (in bytes) of the domain name in the **nam-adr** buffer.

***aggregate-type***
Receives the translated aggregate type.

## Description

The OBJECT ID TO AGGREGATE TYPE routine translates an object identifier to a root aggregate type.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_INVAGGTYP | Invalid aggregate type |

# 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.

## VAX FORMAT

**status = cda$open_converter**

*(standard-item-list ,converter-context ,front-end-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **standard-item-list** | VMS usage: | item_list_2 |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference, array reference |
| **converter-context** | VMS usage: | context |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **front-end-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaOpenConverter**

*(standard_item_list, converter_context, front_end_handle)*

## Argument Information

```
unsigned long CdaOpenConverter(standard_item_list,
                converter_context, front_end_handle)
            unsigned long       *standard_item_list;
            unsigned long       converter_context;
            unsigned long       *front_end_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***standard-item-list***
An item list that identifies the document source and destination, and can also contain options to control processing.

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

*CDA$_INPUT_FRONT_END_DOMAIN*
The parameter is the address and length of a string that specifies the input document domain (either DDIF or DTIF).

*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 Chapter 9.

*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. For more information on the calling sequence for a user *get* routine, see Chapter 9.

*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***
Context value passed as a parameter to the ddif$read_**format** or dtif$read_**format** entry point in the front end.

***front-end-handle***
Receives the handle of the front end that will process the nested content. This handle must be used in all subsequent operations relating to that front end.

## Description

The OPEN CONVERTER routine activates an additional front end to process nested content that is an entire document. The nested content may be in the same format as that of the main document or in a different format.

Processing options that were specified at the main conversion call (either from the command line or by the CONVERT routine) for this document format are automatically retrieved and appended to the standard item list to create a front end item list that is then passed to the front end's main entry point.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_UNSUPFMT | Unsupported document format |

Any error returned by the specific front end.

# OPEN FILE

Opens the specified file for input and validates that its initial contents are valid compound document data. An input stream and a root aggregate are also created.

## VAX FORMAT

**status = 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)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **file-spec-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **file-spec** | VMS usage: | char_string |
| | Data type: | character string |
| | Access: | read only |
| | Mechanism: | by reference |
| **default-file-spec-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

| Argument | Argument Information | |
|---|---|---|
| **default-file-spec** | VMS usage: | char_string |
| | Data type: | character string |
| | Access: | read only |
| | Mechanism: | by reference |
| **alloc-rtn** | VMS usage: | procedure |
| | Data type: | procedure entry mask |
| | Access: | call after stack unwind |
| | Mechanism: | by reference |
| **dealloc-rtn** | VMS usage: | procedure |
| | Data type: | procedure entry mask |
| | Access: | call after stack unwind |
| | Mechanism: | by reference |
| **alloc-dealloc-prm** | VMS usage: | context |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by value |
| **aggregate-type** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **processing-options** | VMS usage: | item_list_2 |
| | Data type: | record |
| | Access: | read only |
| | Mechanism: | by reference, array reference |
| **result-file-spec-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **result-file-spec** | VMS usage: | char_string |
| | Data type: | character string |
| | Access: | write only |
| | Mechanism: | by reference |

| Argument | Argument Information | |
|---|---|---|
| **result-file-ret-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **stream-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **file-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaOpenFile**

*(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)*

## Argument Information

```
unsigned long CdaOpenFile(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)
               unsigned long       file_spec_len;
               unsigned char       *file_spec;
               unsigned long       default_file_spec_len;
               unsigned char       *default_file_spec;
               unsigned long       (*alloc_rtn)();
               unsigned long       (*dealloc_rtn)();
               unsigned long       alloc_dealloc_prm;
               unsigned long       aggregate_type;
               unsigned long       *processing_options;
               unsigned long       result_file_spec_len;
               unsigned char       *result_file_spec;
               unsigned long       *result_file_ret_len;
               unsigned long       *stream_handle;
               unsigned long       *file_handle;
               unsigned long       *root_aggregate_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***file-spec-len***
The length of the string specified by the **file-spec** parameter.

***file-spec***
The file specification.

***default-file-spec-len***
The length (in bytes) of the buffer specified by **default-file-spec**. 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***
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. On ULTRIX systems, the string is appended to the file specification, if the file specification does not already contain a period.

***alloc-rtn***
Address of a memory allocation routine. The calling sequence for an allocation routine is defined in the Description section of this routine. If you specify 0 for this argument, a default memory allocation routine is used. For a description, see Chapter 9.

### *dealloc-rtn*

Address of a memory deallocation routine. The calling sequence for a deallocation routine is defined in the Description section of this routine. If you specify 0 for this argument, a default memory deallocation routine is used. For a description, see Chapter 9.

### *alloc-dealloc-prm*

User context to be passed to the memory allocation and deallocation routines. If the system default memory allocation or deallocation procedure is used, this value is ignored. For a description, see Chapter 9.

### *aggregate-type*

The type of aggregate, expressed as a symbolic constant. The only valid root aggregate types are DDIF$_DDF and DTIF$_DTF.

### *processing-options*

An item list containing options to control processing. 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$_INHERIT_ATTRIBUTES | Inheritance is applied to all document segments. First, if a segment has a type reference that corresponds to a type definition, the attributes of the type are applied to the segment.<br><br>If a segment is the root segment, and a style guide is referenced in the document's header, the definitions and layout from the style guide are applied to the root segment. For the root segment only, standard defined initial values are applied to the attributes of the segment that do not yet have values.<br><br>If the segment is not the root segment, attribute values of its parent segment are applied to the attributes of the segment that do not yet have values. For more information on the inherit attributes processing option, see Section 1.6.1. |
| DDIF$_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$_EVALUATE_CONTENT is specified. For more information on the retain definitions processing option, see Section 1.6.2. |
| DDIF$_EVALUATE_CONTENT | Content reference (DDIF$_CRF) aggregates are replaced with the value of the definition (DDIF$_CTD) they reference. The value of this content definition may be in the document or in an external reference. |

| | |
|---|---|
| | Content for segments with the DDIF$_SGA_COMPUTE_C item present in the segment's attributes (DDIF$_SGA) may be imported from an external reference. If the value of the DDIF$_SGA_COMPUTE_C item is DDIF$K_REMOTE_COMPUTE, the external content is imported and replaces the segment's original content. If the value of the DDIF$_SGA_COMPUTE_C item is DDIF$_K_COPY_COMPUTE, the external content is imported only if the segment has no content. For more information on the evaluate content processing option, see Section 1.6.3. |
| DDIF$_DISCARD_I_SEGMENTS | Segments of the image ( $I ) content category, and any nested segments, are discarded. For more information on the discard image segments processing option, see Section 1.6.4. |
| DDIF$_DISCARD_2D_SEGMENTS | Segments of the graphics ( $2D ) content category, and any nested segments, are discarded. For more information on the discard graphics segments processing option, see Section 1.6.4. |
| DDIF$_DISCARD_T_SEGMENTS | Segments of the text ( $T ) content category, and any nested segments, are discarded. For more information on the discard text segments processing option, see Section 1.6.4. |
| DDIF$_DISCARD_TBL_SEGMENTS | Segments of the table ( $TBL ) content category, and any nested segments, are discarded. For more information on the discard table segments processing option, see Section 1.6.4. |
| DDIF$_DISCARD_PDL_SEGMENTS | Segments of the page description language ( $PDL ) content category, and any nested segments, are discarded. For more information on the discard page descriptions language segments processing option, see Section 1.6.4. |

***result-file-spec-len***
Length of the buffer (in bytes) specified by **result-file-spec**. If you specify 0 for this parameter, the resultant file specification length is not returned.

***result-file-spec***
Receives the resultant 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. On ULTRIX systems, the **file-spec** argument is copied to this buffer.

***result-file-ret-len***
Receives the actual length (in bytes) of the resultant file specification.

***stream-handle***
Receives a value that identifies the newly created stream. This handle must be used in all subsequent operations on that stream.

***file-handle***
Receives a value that identifies the newly opened file. This handle must be used in all subsequent operations on that file.

***root-aggregate-handle***
Receives a value that identifies the newly created root aggregate. 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 initial contents of the file are compound document data. At the same time, this routine also creates an input stream and a root aggregate.

## RETURN VALUES

| Return Value | Description |
|---|---|
| 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

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.

```
     .
     .
     .
/* 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);

    .
    .
    .
```

# OPEN STREAM

Opens a compound document stream for input.

## VAX FORMAT

**status = cda$open_stream**

*(alloc-rtn ,dealloc-rtn ,alloc-dealloc-prm ,get-rtn ,get-prm ,stream-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **alloc-rtn** | VMS usage: | procedure |
| | Data type: | procedure entry mask |
| | Access: | call after stack unwind |
| | Mechanism: | by reference |
| **dealloc-rtn** | VMS usage: | procedure |
| | Data type: | procedure entry mask |
| | Access: | call after stack unwind |
| | Mechanism: | by reference |
| **alloc-dealloc-prm** | VMS usage: | context |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by value |
| **get-rtn** | VMS usage: | procedure |
| | Data type: | procedure entry mask |
| | Access: | call after stack unwind |
| | Mechanism: | by reference |

| Argument | Argument Information | |
|---|---|---|
| **get-prm** | VMS usage: | context |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by value |
| **stream-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaOpenStream**

*(alloc_rtn, dealloc_rtn, alloc_dealloc_prm, get_rtn, get_prm, stream_handle)*

## Argument Information

```
unsigned long CdaOpenStream(alloc_rtn, dealloc_rtn,
                    alloc_dealloc_prm, get_rtn,
                    get_prm, stream_handle)
              unsigned long       (*alloc_rtn)();
              unsigned long       (*dealloc_rtn)();
              unsigned long       alloc_dealloc_prm;
              unsigned long       (*get_rtn)();
              unsigned long       get_prm;
              unsigned long       *stream_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***alloc-rtn***
Address of a memory allocation routine. The calling sequence for an allocation routine is defined in the Description section of this routine. If you specify 0 for this argument, a default memory allocation routine is used. For a description, see Chapter 9.

***dealloc-rtn***
Address of a memory deallocation routine. The calling sequence for a deallocation routine is defined in the Description section of this routine. If you specify 0 for this argument, a default memory deallocation routine is used. For a description, see Chapter 9.

***alloc-dealloc-prm***
User context to be passed to the memory allocation and deallocation routines. If the system default memory allocation or deallocation routine is used, this parameter is ignored. For a description, see Chapter 9.

***get-rtn***
Address of a stream *get* routine. The calling sequence for a *get* routine is defined in Chapter 9. If you specify 0 for this argument on VMS systems, a default get-rtn is used. On ULTRIX systems, you must supply both get-rtn and get-prm; there is no default. 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***
User context to be passed to the stream *get* routine. If the VMS system default *get* routine is used, the value must be a pointer to a RAB. On ULTRIX systems, if you specify a value for the **get-rtn**, you must supply a value other than 0 for the **get-prm** argument. For a description, see Chapter 9.

***stream-handle***
Receives a value that identifies the newly created stream. 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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

Any error returned by the memory allocation routines.

# OPEN TEXT FILE

Opens a standard text file for input.

## VAX FORMAT

**status = 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)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **file-spec-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **file-spec** | VMS usage: | char_string |
| | Data type: | character string |
| | Access: | read only |
| | Mechanism: | by reference |
| **default-file-spec-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **default-file-spec** | VMS usage: | char_string |
| | Data type: | character string |
| | Access: | read only |
| | Mechanism: | by reference |

| Argument | Argument Information | |
|---|---|---|
| **result-file-spec-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **result-file-spec** | VMS usage: | char_string |
| | Data type: | character string |
| | Access: | write only |
| | Mechanism: | by reference |
| **result-file-ret-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **text-file-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaOpenTextFile**

*(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)*

## Argument Information

```
unsigned long CdaOpenTextFile(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)
              unsigned long         file_spec_len;
              unsigned char         *file_spec;
              unsigned long         default_file_spec_len;
              unsigned char         *default_file_spec;
              unsigned long         result_file_spec_len;
              unsigned char         *result_file_spec;
              unsigned long         *result_file_ret_len;
              unsigned long         *text_file_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***file-spec-len***
Length (in bytes) of the string specified by the **file-spec** argument.

***file-spec***
File specification of the text file to be opened for input.

***default-file-spec-len***
Length (in bytes) of the string specified by **default-file-spec**. If you specify 0 for this parameter, no default file specification is used.

***default-file-spec***
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. On ULTRIX systems, the string is appended to the file specification if the file specification does not already contain a period.

***result-file-spec-len***
Length (in bytes) of the buffer specified by **result-file-spec**. If you specify 0 for this parameter, the length of the resultant file specification is not returned.

***result-file-spec***
Receives the resultant file specification. This file specification is the result of a VMS RMS $OPEN operation. On ULTRIX systems, the file specification is copied to this buffer. If you specify 0 for this parameter, a resultant file specification is not returned.

***result-file-ret-len***
Receives the actual length (in bytes) of the resultant file specification.

***text-file-handle***
Receives the handle of the text file. 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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

Any error returned by the memory allocation routines.

Any error returned by the file routines.

# PRUNE AGGREGATE

Removes the next sequential document content aggregate from an existing in-memory compound document, and returns its handle and type.

## VAX FORMAT

**status = cda$prune_aggregate**

*(root-aggregate-handle ,aggregate-handle ,aggregate-type)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **aggregate-type** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

# PRUNE AGGREGATE

## C FORMAT

**status = CdaPruneAggregate**

*(root_aggregate_handle, aggregate_handle, aggregate_type)*

## Argument Information

```
unsigned long CdaPruneAggregate(root_aggregate_handle,
                aggregate_handle, aggregate_type)
              unsigned long      root_aggregate_handle;
              unsigned long      *aggregate_handle;
              unsigned long      *aggregate_type;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate associated with the aggregate to be removed. 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 the aggregates, you cannot specify the same root aggregate handle again when calling the PRUNE AGGREGATE routine.

***aggregate-handle***
Receives the handle of the removed aggregate. This handle must be used in all subsequent operations on that aggregate.

***aggregate-type***
Receives the 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 primary aggregate from an existing in-memory compound document and returns the aggregate identifier and type. Primary aggregates, also known as "top-level" aggregates, include all the document content aggregates and the DDIF$_DHD, DDIF$_DSC, and DDIF$_EOS aggregates. A front end should invoke this routine from the *get-aggregate* entry point module in cases where the front end builds an entire compound document in memory before returning its content.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_ENDOFDOC | End of document |

# PRUNE POSITION

Returns the position in and size of an in-memory document.

## VAX FORMAT

**status = cda$prune_position**

*(root-aggregate-handle ,file-position ,file-size)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **file-position** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **file-size** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaPrunePosition**

*(root_aggregate_handle, file_position, file_size)*

## Argument Information

```
unsigned long CdaPrunePosition(root_aggregate_handle,
                file_position, file_size)
              unsigned long      root_aggregate_handle;
              unsigned long      *file_position;
              unsigned long      *file_size;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate associated with the in-memory document. The handle is returned by a call to either the OPEN FILE routine or the CREATE ROOT AGGREGATE routine.

***file-position***
Receives the current position (in bytes) as measured from the start of the document being processed.

***file-size***
Receives the total size (in bytes) of the in-memory document being processed.

## 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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

# PUT AGGREGATE

Writes one or more aggregates to a specified stream.

## VAX FORMAT

**status = cda$put_aggregate**

*(root-aggregate-handle ,stream-handle ,aggregate-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **stream-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaPutAggregate**

*(root_aggregate_handle, stream_handle, aggregate_handle)*

## Argument Information

```
unsigned long CdaPutAggregate(root_aggregate_handle,
             stream_handle, aggregate_handle)
             unsigned long       root_aggregate_handle;
             unsigned long       stream_handle;
             unsigned long       aggregate_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate associated with the aggregate to be written. 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***
Identifier of the stream to which the aggregate is to be written. This handle is returned by a call to either the CREATE FILE routine or the CREATE STREAM routine.

***aggregate-handle***
Identifier of the aggregate to be written.

## 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 must be one of the following primary DDIF or DTIF aggregates:

| 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$_EOS | End of segment |
| DTIF$_HDR | Document header |
| DTIF$_CLD | Cell data |
| DTIF$_DSC | Document descriptor |
| DTIF$_TBL | Table definition |
| DTIF$_ROW | Row definition |

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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_INVAGGTYP | Invalid aggregate type |
| CDA$_INVDOC | Invalid document content |

Any error returned by the file routines.

# PUT DOCUMENT

Writes an entire document to the specified stream. The document is not changed by this operation.

## VAX FORMAT

**status = cda$put_document**

*(root-aggregate-handle ,stream-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **stream-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaPutDocument**

*(root_aggregate_handle, stream_handle)*

# PUT DOCUMENT

## Argument Information

```
unsigned long CdaPutDocument(root_aggregate_handle,
                  stream_handle)
            unsigned long       root_aggregate_handle;
            unsigned long       stream_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate associated with the document to be written. 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***
Identifier of the stream to which the document is to be written. 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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_INVDOC | Invalid document content |

Any error returned by the file routines.

# READ TEXT FILE

Reads a line from a standard text file.

## VAX FORMAT

**status = cda$read_text_file**

*(text-file-handle ,buffer-length ,buffer-address)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **text-file-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **buffer-length** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **buffer-address** | VMS usage: | address |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaReadTextFile**

*(text_file_handle, buffer_length, buffer_address)*

## Argument Information

```
unsigned long CdaReadTextFile(text_file_handle,
                buffer_length, buffer_address)
               unsigned long        text_file_handle;
               unsigned long        *buffer_length;
               unsigned char        *buffer_address;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***text-file-handle***
Identifier of the text file from which the line is to be read. This handle is returned by a call to the OPEN TEXT FILE routine.

***buffer-length***
Receives the length (in bytes) of the line that is read.

***buffer-address***
Receives the address of the line that is read. No trailing record delimiter is present. On ULTRIX, **buffer-address** receives the address of the line up to, but not including, the new-line indicator.

## 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. On ULTRIX systems, the line is delimited by a new-line character.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_ENDOFDOC | End of document |

Any error returned by the file routines.

# REMOVE AGGREGATE

Removes an aggregate from a sequence. The aggregate is not deleted. If the specified aggregate is not part of a sequence and has no parent aggregate, no operation is performed.

## VAX FORMAT

**status = cda$remove_aggregate**
*(aggregate-handle)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaRemoveAggregate**
*(aggregate_handle)*

## Argument Information

```
unsigned long CdaRemoveAggregate(aggregate_handle)
               unsigned long     aggregate_handle;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***aggregate-handle***
Identifier of the aggregate to be removed from the sequence.

## Description

The REMOVE AGGREGATE routine removes an aggregate that is part of a sequence from that sequence. The aggregate is not deleted. If the aggregate is not part of a sequence and has no parent aggregate, no action is performed.

**NOTE**

Do not attempt to use the REMOVE AGGREGATE routine to remove the only aggregate from a single-aggregate sequence, or to remove an aggregate from its parent when the corresponding aggregate-valued item of the parent is not defined to be a sequence of aggregates.

Although current implementation of the REMOVE AGGREGATE routine allows removing the only aggregate from a single-aggregate sequence, and even allows removing an aggregate from its parent when the corresponding aggregate-valued item of the parent is not defined to be a sequence of aggregates, the use of the REMOVE AGGREGATE routine in this manner is not supported and may leave the aggregate data structures in an inconsistent state. This use of the REMOVE AGGREGATE routine may be prevented (causing an error status to be returned) in a future release of the CDA toolkit.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

# 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.

## VAX FORMAT

**status = cda$store_item**

*(root-aggregate-handle ,aggregate-handle ,aggregate-item ,buf-len ,buf-adr [,aggregate-index] [,add-info])*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **root-aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-item** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

| Argument | Argument Information | |
|---|---|---|
| **buf-len** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **buf-adr** | VMS usage: | vector_byte_unsigned |
| | Data type: | byte (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference, array reference |
| **aggregate-index** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **add-info** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaStoreItem**

*(root_aggregate_handle, aggregate_handle, aggregate_item, buf_len, buf_adr, aggregate_index, add_info)*

## Argument Information

```
unsigned long CdaStoreItem(root_aggregate_handle,
                    aggregate_handle, aggregate_item,
                    buf_len, buf_adr, aggregate_index,
                    add_info)
              unsigned long       root_aggregate_handle;
              unsigned long       aggregate_handle;
              unsigned long       aggregate_item;
              unsigned long       buf_len;
              unsigned char       *buf_adr;
              unsigned long       aggregate_index;
              unsigned long       add_info;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***root-aggregate-handle***
Identifier of the root aggregate with which the aggregate containing the item is associated. 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***
Identifier of the aggregate into which the item is written.

***aggregate-item***
Identifying code of the item, expressed as a symbolic constant. The DDIF aggregate item symbolic constants are defined in the file ddif$def.h on VMS systems and in the file ddif_def.h on ULTRIX systems and are described in Chapter 4. The DTIF aggregate type symbolic codes are defined in the file dtif$def.h on VMS systems and in the file dtif_def.h on ULTRIX systems and are described in Chapter 5.

A user context item named DDIF$_USER_CONTEXT for DDIF aggregates and DTIF$_USER_CONTEXT for DTIF aggregates is available within every aggregate. This item is a longword that can be used by the application for any purpose.

***buf-len***
Length (in bytes) of the buffer specified by the **buf-adr** argument.

***buf-adr***
Buffer containing the item's value.

***aggregate-index***
Index of the item (relative to 0). 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***
Data type-specific modifier for the data types character string, string with **add-info**, general floating-point, and scaled integer. For data types other than character string, string with **add-info**, general floating-point, and scaled integer, 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 scaled integer, the **add-info** parameter receives an integer scaling factor. 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. For the data type general floating-point, the **add-info** parameter contains a value that identifies the format of the floating-point value supplied in **buf-adr**. For **add-info** values for the general floating-point type, see Table 1–1. 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 files ddif$def.h and dtif$def.h on VMS systems and in the files ddif_def.h and dtif_def.h on ULTRIX systems, and are described in Chapter 4 and in Chapters 5 through 7. 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 specified must not exceed one more than the maximum index of the previously stored indexed 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.

## RETURN VALUES

| Return Value | Description |
|---|---|
| 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. |

| Return Value | Description |
|---|---|
| CDA$_VARVALUE | Variant value is undefined. |
| CDA$_INVINSERT | Aggregate already in a sequence. |
| CDA$_INVBUFLEN | Invalid buffer length. |

## Examples

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.

```
      .
      .
      .
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 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.

```
     .
     .
     .
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);

      .
      .
      .
```

# WRITE TEXT FILE

Writes a line of text to a standard text file.

## VAX FORMAT

**status = cda$write_text_file**

*(text-file-handle ,buffer-length ,buffer-address)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | VMS usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **text-file-handle** | VMS usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **buffer-length** | VMS usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **buffer-address** | VMS usage: | char_string |
| | Data type: | character string |
| | Access: | read only |
| | Mechanism: | by reference |

## C FORMAT

**status = CdaWriteTextFile**

*(text_file_handle, buffer_length, buffer_address)*

## Argument Information

```
unsigned long CdaWriteTextFile(text_file_handle,
                    buffer_length, buffer_address)
              unsigned long       text_file_handle;
              unsigned long       buffer_length;
              unsigned char       *buffer_address;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***text-file-handle***
Identifier of the text file to which the line is written. This handle is returned by a call to the CREATE TEXT FILE routine.

***buffer-length***
Length (in bytes) of the buffer specified by the **buffer-address** argument.

***buffer-address***
The line to be written to the text file.

## 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. On ULTRIX systems, the written line is followed by a new-line character.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

Any error returned by the file routines.

# Chapter 9

# User-Defined Routines

The chapter describes the user-defined routines used to write both CDA-conforming applications and front and back ends. You can supply these routines to modify the operation of the CDA Toolkit routines. For example, the GET AGGREGATE routine, by default, calls a CDA Toolkit *get* routine. However, you may provide your own *get* routine using the format described in this chapter.

Each routine description includes the following information:

- A routine definition that each application must name according to its operating system-specific format
- Descriptions of each routine argument
- A description of the routine itself
- A list of possible values returned by each routine argument

**NOTE**

> The entry points and conventions defined throughout this reference section must be followed on both VMS and ULTRIX systems in order for all front and back ends to work properly with the CDA Converter Kernel.

If you are programming in Ada, please refer to the *Guide to Applications Programming* for information on Ada programming with DECwindows.

# *Allocate/Deallocate* Routines

Are the specification of the calling standard for two optional user-supplied routines used to perform memory allocation and deallocation. The address of these routines can be passed to the CREATE FILE, CREATE ROOT AGGREGATE, CREATE STREAM, OPEN STREAM, or OPEN FILE routine. If specified, these allocation and deallocation routines will be used throughout the CDA Toolkit to allocate and deallocate memory.

## FORMAT

**status = user-rtn** *(num-bytes ,base-adr ,alloc-dealloc-prm)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | Usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **num-bytes** | Usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **base-adr** | Usage: | address |
| | Data type: | longword (unsigned) |
| | Access: | read only or write only |
| | Mechanism: | by reference |
| **alloc-dealloc-prm** | Usage: | user_arg |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by value |

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***num-bytes***
The number of bytes to allocate or deallocate. The value of **num-bytes** must be greater than zero.

***base-adr***
Virtual address of the first byte of memory to be allocated or deallocated. (This argument is write-only for an *allocate* routine, and read-only for a *deallocate* routine.)

***alloc-dealloc-prm***
User context argument.

## Description

The *allocate / deallocate* routines are the specification of the calling standard for two optional user-supplied routines used to perform memory allocation and deallocation. The address of these routines can be passed to the CREATE FILE, CREATE ROOT AGGREGATE, CREATE STREAM, OPEN STREAM, or OPEN FILE routine. If specified, these allocation and deallocation routines will be used throughout the CDA Toolkit to allocate and deallocate memory.

The **alloc-dealloc-prm** argument is passed through these CDA routines to the user-supplied routine. For example, the **alloc-dealloc-prm** argument must be supplied to the CREATE FILE routine, which will then pass it to the user-supplied *allocate* routine.

## RETURN VALUES

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.

# *Flush* Routine

Is a specification of the calling standard for an optional user-supplied routine. The address of a routine that meets this specification can be passed to the FLUSH STREAM routine, which will use the routine to force the writing of the user's buffer.

## FORMAT

**status = flush-rtn** *(flush-prm)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | Usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **flush-prm** | Usage: | user_arg |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by value |

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***flush-prm***
User context argument.

## Description

The *flush* routine is a specification of the calling standard for an optional user-supplied routine. The address of a routine that meets this specification can be passed to the FLUSH STREAM routine, which will use the routine to force the writing of the user's buffer.

The user-supplied *flush* routine is only neccessary when a user-supplied *put* routine (using buffered output) has been specified in the call to the CREATE STREAM routine.

## RETURN VALUES

The user-defined *flush* routine must return a value that is one of the error status codes named by the CDA Toolkit (such as CDA$_INVDOC), by VMS RMS, or that is application-defined.

If the first bit of the longword returned by this routine is set to 1, the return status is successful. However, if the first bit of the longword returned by this routine is set to 0, the return status is unsuccessful.

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.

# *Get* Routine

Is a specification of the calling standard for an optional user-supplied routine. The address of a routine that meets this specification can be passed to the CONVERT routine or to the OPEN STREAM routine.

## FORMAT

**status = get-rtn** *(get-prm ,num-bytes ,buf-adr)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | Usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **get-prm** | Usage: | user_arg |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by value |
| **num-bytes** | Usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **buf-adr** | Usage: | address |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***get-prm***
User context argument.

***num-bytes***
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***
Receives the address of an unsigned longword that receives the buffer address.

## Description

The *get* routine is a specification of the calling standard for an optional user-supplied routine. The address of a routine that meets this specification can be passed to the CONVERT routine or to the OPEN STREAM routine.

For the CONVERT routine, this address will be passed to a front end converter, which may then use the specified routine to read input data. Refer to the example for the CONVERT routine in Chapter 8.

For the OPEN STREAM routine, this address will be stored for use by the GET AGGREGATE routine when reading input data.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |
| CDA$_ENDOFDOC | End of document |

# *Get-Position* Routine

Is a specification of the calling standard for an optional user-supplied routine. The address of a routine that meets this specification can be passed to the CONVERT routine. This user-supplied routine must provide position information to allow the application reader or converter 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.)

## FORMAT

**status = get-pos-rtn** *(stream-prm ,stream-size)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | Usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **stream-prm** | Usage: | user_arg |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by value |
| **stream-size** | Usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***stream-prm***
User context argument.

***stream-size***
Receives the number of bytes contained in the buffer. The **stream-size** 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.

## Description

The *get-position* routine is a specification of the calling standard for an optional user-supplied routine. The address of a routine that meets this specification can be passed to the CONVERT routine. This user-supplied routine must provide position information to allow the application reader or converter 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.)

## RETURN VALUES

The user-defined *get-position* routine can return a value that is one of the error status codes named by the CDA Toolkit (such as CDA$_INVDOC), by VMS RMS, or one that is application-defined.

If the first bit (bit 0) of the longword returned by this routine is set to 1, the return status is successful. However, if the first bit of the longword returned by this routine is set to 0, the return status is unsuccessful.

# *Put* Routine

Is a specification of the calling standard for an optional user-supplied routine. The address of a routine that meets this specification can be passed to the CONVERT routine or to the CREATE STREAM routine.

## FORMAT

**status = put-rtn** *(put-prm ,num-bytes ,buf-adr ,next-buf-len ,next-buf-adr)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | Usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **put-prm** | Usage: | user_arg |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by value |
| **num-bytes** | Usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **buf-adr** | Usage: | vector_byte_unsigned |
| | Data type: | byte (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference, array reference |
| **next-buf-len** | Usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

| Argument | Argument Information | |
|---|---|---|
| **next-buf-adr** | Usage: | address |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***put-prm***
User context argument.

***num-bytes***
Number of bytes contained in the buffer. The **num-bytes** argument is the address of an unsigned longword that contains this value.

***buf-adr***
Address of the buffer. The **buf-adr** argument is the address of an array of unsigned bytes.

***next-buf-len***
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***
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.

## Description

The *put* routine is a specification of the calling standard for an optional user-supplied routine. The address of a routine that meets this specification can be passed to the CONVERT routine or to the CREATE STREAM routine.

For the CONVERT routine, this address will be passed to a back end converter, which may then use the specified routine to write output data. Refer to the example for the CONVERT routine in Chapter 8.

For the CREATE STREAM routine, this address will be stored for use by the PUT AGGREGATE routine when writing output data.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

The user-defined *put* routine always returns a status CDA$_NORMAL in an unsigned longword integer.

The user-defined *put* routine can also return a value that is one of the error status codes named by the CDA Toolkit (such as CDA$_INVDOC) or that is application-defined.

If the first bit of the longword returned by this routine is set to 1, the return status is successful. However, if the first bit of the longword returned by this routine is set to 0, the return status is unsuccessful.

# Chapter 10

# CDA Toolkit Example Program

This chapter illustrates a sample program, written in VAX C, that uses the CDA Toolkit to create a DDIF document. Example 10–1 contains comments where necessary, and Example 10–2 illustrates the analysis output of the DDIF document created by the program. The callouts in this example correspond to the callouts in Example 10–2. For example, if a callout corresponds to a call to the CREATE ROOT AGGREGATE routine in Example 10–1, the callout in Example 10–2 identifies the beginning of the document root aggregate created by that call.

**Example 10–1: Sample CDA Toolkit Program**

```
#ifdef vms
#include <ddif$def.h>          /* Include DDIF keyword definitions.        */
#include <cda$def.h>           /* Include CDA Toolkit keyword definitions. */
#else
#include <ddif_def.h>          /* Include DDIF keyword definitions.        */
#include <cda_def.h>           /* Include CDA Toolkit keyword definitions. */
#endif

#define FAILURE(x)     (((x) & 1) == 0)
#define MAX_POINTS 4

/*
** Subroutines to generate frequently-used aggregates.
*/

extern unsigned long create_hrd_dir( );
extern unsigned long create_gtx( );
unsigned long   poly_points[MAX_POINTS][2] =
                        {
                        { 500, 500 },
                        { 2500, 2000 },
                        { 3500, 2000 },
                        { 5500, 500 }
                        };
unsigned long   aggregate_type;
unsigned long   aggregate_item;
unsigned long   aggregate_index;
unsigned long   add_info;
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;
```

(continued on next page)

```
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;

/* 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;

/* Data for the polyline and Bezier curve. */

unsigned long   i;

/* 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 long       erf_data_type[] =
                        {1,3,12,1011,1,3,1};
unsigned long       erf_data_type_length = sizeof(erf_data_type);

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       erf_descriptor_name[] = "Style Guide";
unsigned long       erf_descriptor_name_length = sizeof( erf_descriptor_name ) - 1;

unsigned char       erf_label_name[] = "defstyle";
unsigned long       erf_label_name_length = sizeof( erf_label_name ) - 1;

unsigned char       erf_label_type[] = "$STYLE";
unsigned long       erf_label_type_length = sizeof( erf_label_type ) - 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;
```

(continued on next page)

## Example 10–1 (Cont.): Sample CDA Toolkit Program

```
unsigned char   sga_content_category[] = "$2D";
unsigned long   sga_content_category_length_2 =
                        sizeof( sga_content_category ) - 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 char   gtx_content_2[] = "The third line of the example text will \
begin on a new line.";

unsigned char        gtx_content_3[] = "The fourth line of the example text will be \
separated from the previous lines by a blank line, and will be the \
last text on the first page.";

unsigned char        para_seg_type[] = "PARA";
unsigned long        para_seg_type_length = sizeof( para_seg_type ) - 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;
main()
{

    printf("Creating in-memory DDIF structure...\n" );
```

(continued on next page)

```
/*
******************************************************************************
**
**   The overall structure (excluding hard directives) is as follows:
**
**                      DDF
**                (Root Aggregate)
**
**                 /      |      \
**              DSC      DHD      SEG
**        (Descriptor) (Header) (Segment)
**
**                                 |
**                                SEG
**
**                                 |
**                                TXT - GTX - SEG - GTX - SEG - GTX - SEG - GTX
**
**                                            |           |           |
**                                           LIN         BEZ         SEG
**                                                                    |
**                                                                   ARC
**
******************************************************************************
*/

    /*
    ** 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 );

/*
******************************************************************************
**
**   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 );
```

(continued on next page)

```
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 );


/*
** 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.
*/
```

(continued on next page)

```
    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 );

    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 );

    /*
    ** Add the style guide reference.
    */

    ahs_index++;
    aggregate_type = DDIF$_ERF;
    status = cda$create_aggregate( &root_aggregate_handle,  ❽
                                   &aggregate_type,
                                   &aggregate_handle_stack[ahs_index] );
    if( FAILURE( status ) )
        return ( status );
```

(continued on next page)

```
aggregate_item = DDIF$_DHD_EXTERNAL_REFERENCES;
status = cda$store_item( &root_aggregate_handle,    ❾
                         &aggregate_handle_stack[ahs_index-1],
                         &aggregate_item,
                         &aggregate_handle_length,
                         &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_ERF_DATA_TYPE;
status = cda$store_item( &root_aggregate_handle,   ❿
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &erf_data_type_length,
                          erf_data_type );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_ERF_DESCRIPTOR;
aggregate_index = 0;
add_info = CDA$K_ISO_LATIN1;
status = cda$store_item( &root_aggregate_handle,   ⓫
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &erf_descriptor_name_length,
                          erf_descriptor_name,
                         &aggregate_index,
                         &add_info );
if( FAILURE( status ) )
    return ( status );


aggregate_index = 0;
add_info = CDA$K_ISO_LATIN1;
aggregate_item = DDIF$_ERF_LABEL;
status = cda$store_item( &root_aggregate_handle, ⓬
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &erf_label_name_length,
                          erf_label_name,
                         &aggregate_index,
                         &add_info );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_ERF_LABEL_TYPE;
add_info = DDIF$K_STYLE_LABEL_TYPE;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &erf_label_type_length,
                          erf_label_type,
                         0,
                         &add_info);

if( FAILURE( status ) )
    return ( status );
```

(continued on next page)

```
aggregate_item = DDIF$_ERF_CONTROL;
integer_value = DDIF$K_NO_COPY_REFERENCE;
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--;


/*
** 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,     ⓭
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value,
                         &aggregate_index );
if( FAILURE( status ) )
    return ( status );


aggregate_item = DDIF$_DHD_LANGUAGES;
aggregate_index = 0;
status = cda$store_item( &root_aggregate_handle,   ⓮
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &dhd_languages_length_1,
                          dhd_languages_1,
                         &aggregate_index );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_DHD_LANGUAGES_C;
aggregate_index = 1;
status = cda$store_item( &root_aggregate_handle,    ⓯
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value,
                         &aggregate_index );
if( FAILURE( status ) )
    return ( status );
```

(continued on next page)

```
    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,   ⓰
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &dhd_languages_length_2,
                              dhd_languages_2,
                             &aggregate_index,
                             &add_info );
    if( FAILURE( status ) )
        return ( status );

    /*
    ** Add the DHD_STYLE_GUIDE item, defining it to point to the
    ** first external reference, which was defined above as the
    ** style-guide file.
    */

    aggregate_item = DDIF$_DHD_STYLE_GUIDE;
    integer_value = 1;
    status = cda$store_item( &root_aggregate_handle,
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &integer_length,
                             &integer_value);
    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,   ⓱
                                   &aggregate_type,
                                   &aggregate_handle_stack[ahs_index] );
    if( FAILURE( status ) )
        return ( status );
```

(continued on next page)

## Example 10–1 (Cont.): Sample CDA Toolkit Program

```
aggregate_item = DDIF$_DDF_CONTENT;
status = cda$store_item( &root_aggregate_handle,    ⓲
                         &root_aggregate_handle,
                         &aggregate_item,
                         &aggregate_handle_length,
                         &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

ahs_index++;
aggregate_type = DDIF$_SEG;
status = cda$create_aggregate( &root_aggregate_handle, ⓳
                               &aggregate_type,
                               &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/* Store into this segment. */

aggregate_item = DDIF$_SEG_CONTENT;
status = cda$store_item( &root_aggregate_handle,   ⓴
                         &aggregate_handle_stack[ahs_index-1],
                         &aggregate_item,
                         &aggregate_handle_length,
                         &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/*
** Store the segment type ("PARA"). PARA is defined in the default
** style guide.
*/

aggregate_item = DDIF$_SEG_SEGMENT_TYPE;
status = cda$store_item( &root_aggregate_handle,   ㉑
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &para_seg_type_length,
                          para_seg_type );
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,   ㉒
                               &aggregate_type,
                               &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );
```

(continued on next page)

```
aggregate_item = DDIF$_SEG_SPECIFIC_ATTRIBUTES;
status = cda$store_item( &root_aggregate_handle, ㉓
                         &aggregate_handle_stack[ahs_index-1],
                         &aggregate_item,
                         &aggregate_handle_length,
                         &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

/*
** 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,  ㉔
                               &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,  ㉕
                         &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,  ㉖
                         &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, ㉗
                               &aggregate_type,
                               &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );
```

(continued on next page)

```
aggregate_item = DDIF$_TYD_ATTRIBUTES;
status = cda$store_item( &root_aggregate_handle,  ㉘
                         &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.
*/

aggregate_item = DDIF$_SGA_CONTENT_CATEGORY;
aggregate_index = 0;
add_info = DDIF$K_2D_CATEGORY;
status = cda$store_item( &root_aggregate_handle,  ㉙
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &sga_content_category_length_2,
                          sga_content_category,
                         &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,  ㉚
                         &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;
aggregate_index = 0;
status = cda$store_item( &root_aggregate_handle,  ㉛
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value,
                         &aggregate_index );
if( FAILURE( status ) )
    return ( status );
```

**(continued on next page)**

**Example 10–1 (Cont.): Sample CDA Toolkit Program**

```
aggregate_item = DDIF$_SGA_FRM_BOX_LL_X;
aggregate_index = 0;
integer_value = 0;
status = cda$store_item( &root_aggregate_handle,  ㉜
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value,
                         &aggregate_index );
if( FAILURE( status ) )
    return ( status );


aggregate_item = DDIF$_SGA_FRM_BOX_LL_Y_C;
integer_value = DDIF$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 );
```

(continued on next page)

```
    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 );


    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 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.)
    */
```

(continued on next page)

```
ahs_index++;
aggregate_type = DDIF$_TXT;
status = cda$create_aggregate( &root_aggregate_handle,    ❸❸
                               &aggregate_type,
                               &aggregate_handle_stack[ahs_index] );
if( FAILURE( status ) )
    return ( status );

aggregate_item = DDIF$_SEG_CONTENT;
status = cda$store_item( &root_aggregate_handle,   ❸❹
                         &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.
*/

aggregate_item = DDIF$_TXT_CONTENT;
status = cda$store_item( &root_aggregate_handle,    ❸❺
                         &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,    ❸❻
                          &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,      ❸❼
                      &previous_aggregate_handle,
                      &aggregate_handle_stack[ahs_index],
                       gtx_content_1 );
if( FAILURE( status ) )
    return ( status );

/* Insert a new-line (hard) directive to force a new line. */
```

(continued on next page)

```
previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_hrd_dir ( &root_aggregate_handle,   ㊳
                          &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,
                      &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. */

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_hrd_dir ( &root_aggregate_handle,
                          &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_hrd_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,
                      &previous_aggregate_handle,
                      &aggregate_handle_stack[ahs_index],
                       gtx_content_3 );
if( FAILURE( status ) )
    return ( status );

/* Insert a new-page (hard) directive. */

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_hrd_dir ( &root_aggregate_handle,
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_PAGE );
if( FAILURE( status ) )
    return ( status );
```

(continued on next page)

```
    /* Insert next general-text line. */

    previous_aggregate_handle = aggregate_handle_stack[ahs_index];
    status = create_gtx ( &root_aggregate_handle,
                          &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 );


    /*
    ** Create a new segment aggregate in which to define the polyline, and
    ** insert after the previous aggregate.
    */

    previous_aggregate_handle = aggregate_handle_stack[ahs_index];
    aggregate_type = DDIF$_SEG;
    status = cda$create_aggregate( &root_aggregate_handle,      (39)
                                   &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,      (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. */
```

**(continued on next page)**

```
ahs_index++;
aggregate_type = DDIF$_LIN;
status = cda$create_aggregate( &root_aggregate_handle,     ❹❶
                               &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,      ❹❷
                         &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. */

aggregate_item = DDIF$_LIN_FLAGS;
integer_value = 0x1;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value );
if( FAILURE( status ) )
    return ( status );

/* Store the Line Pattern bit string into the Polyline aggregate. */

aggregate_item = DDIF$_LIN_DRAW_PATTERN;
integer_value = 0xF;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_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$_LIN_PATH_C;
    integer_value = DDIF$K_VALUE_CONSTANT;
    aggregate_index = i * 2;
    status = cda$store_item( &root_aggregate_handle,      ❹❸
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &integer_length,
                             &integer_value,
                             &aggregate_index );
    if( FAILURE( status ) )
        return ( status );
```

**(continued on next page)**

```
    /* Store the x-coordinate integer value in the polyline path array. */

    aggregate_item = DDIF$_LIN_PATH;
    integer_value = poly_points[i][0];
    aggregate_index = i * 2;
    status = cda$store_item( &root_aggregate_handle,     ❹❹
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &integer_length,
                             &integer_value,
                             &aggregate_index );
    if( FAILURE( status ) )
        return ( status );

    /*
    ** Now store the y-coordinate for each point.
    */

    aggregate_item = DDIF$_LIN_PATH_C;
    integer_value = DDIF$K_VALUE_CONSTANT;
    aggregate_index = ((2 * i) + 1 );
    status = cda$store_item( &root_aggregate_handle,     ❹❺
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &integer_length,
                             &integer_value,
                             &aggregate_index );
    if( FAILURE( status ) )
        return ( status );

    aggregate_item = DDIF$_LIN_PATH;
    integer_value = poly_points[i][1];
    aggregate_index = ((2 * i) + 1 );
    status = cda$store_item( &root_aggregate_handle,     ❹❻
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &integer_length,
                             &integer_value,
                             &aggregate_index );
    if( FAILURE( status ) )
        return ( status );
    }; /* End of "for" loop */

ahs_index--; /* End of pline. */                        ❹❼

/* Insert two new-line directives to cause a skipped line. */

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_hrd_dir ( &root_aggregate_handle,     ❹❽
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
if( FAILURE( status ) )
    return ( status );
```

**(continued on next page)**

```
previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_hrd_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,  49
                      &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. */

previous_aggregate_handle = aggregate_handle_stack[ahs_index];
status = create_hrd_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_hrd_dir ( &root_aggregate_handle,
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
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_hrd_dir ( &root_aggregate_handle,
                          &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_hrd_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.
*/
```

(continued on next page)

## Example 10–1 (Cont.): Sample CDA Toolkit Program

```
previous_aggregate_handle = aggregate_handle_stack[ahs_index];
aggregate_type = DDIF$_SEG;
status = cda$create_aggregate( &root_aggregate_handle,  ❺❶
                               &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,  ❺❷
                         &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. */
aggregate_type = DDIF$_BEZ;
previous_aggregate_handle = aggregate_handle_stack[ahs_index];
ahs_index++;
status = cda$create_aggregate( &root_aggregate_handle, ❺❸
                               &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,   ❺❹
                         &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)

```
/* Store the Flags item into the Bezier Curve aggregate. */

aggregate_item = DDIF$_BEZ_FLAGS;
integer_value = 0x1;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_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$_BEZ_PATH_C;
    integer_value = DDIF$K_VALUE_CONSTANT;
    aggregate_index = i * 2;
    status = cda$store_item( &root_aggregate_handle,     ❺❺
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &integer_length,
                             &integer_value,
                             &aggregate_index );
    if( FAILURE( status ) )
        return ( status );

    /* Store the x-coordinate integer value in the polyline path array. */

    aggregate_item = DDIF$_BEZ_PATH;
    integer_value = poly_points[i][0];
    aggregate_index = i * 2;
    status = cda$store_item( &root_aggregate_handle, ❺❻
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &integer_length,
                             &integer_value,
                             &aggregate_index );
    if( FAILURE( status ) )
        return ( status );

    /*
    ** Now store the y-coordinate for each point.
    */

    aggregate_item = DDIF$_BEZ_PATH_C;
    integer_value = DDIF$K_VALUE_CONSTANT;
    aggregate_index = ((2 * i) + 1 );
    status = cda$store_item( &root_aggregate_handle,     ❺❼
                             &aggregate_handle_stack[ahs_index],
                             &aggregate_item,
                             &integer_length,
                             &integer_value,
                             &aggregate_index );
    if( FAILURE( status ) )
        return ( status );
```

(continued on next page)

```
        aggregate_item = DDIF$_BEZ_PATH;
        integer_value = poly_points[i][1];
        aggregate_index = ((2 * i) + 1 );
        status = cda$store_item( &root_aggregate_handle,    58
                                 &aggregate_handle_stack[ahs_index],
                                 &aggregate_item,
                                 &integer_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_hrd_dir ( &root_aggregate_handle,    59
                              &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_hrd_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_hrd_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_hrd_dir ( &root_aggregate_handle,
                              &previous_aggregate_handle,
                              &aggregate_handle_stack[ahs_index],
                               DDIF$K_DIR_NEW_LINE );
    if( FAILURE( status ) )
        return ( status );
```

(continued on next page)

```
/*
** 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. */

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 );
```

**(continued on next page)**

```
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 );

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 );
```

**(continued on next page)**

```
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 );

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 line attributes */        68
```

(continued on next page)

```
/* 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, ⓳
                               &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,  ⓴
                         &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. */

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$_ARC_FLAGS;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_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$_ARC_CENTER_X_C;
integer_value = DDIF$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 );
```

(continued on next page)

```
/* Store an integer value for the arc center x-coordinate. */

aggregate_item = DDIF$_ARC_CENTER_X;
integer_value = 3000;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_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$_ARC_CENTER_Y_C;
integer_value = DDIF$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 );

/* Store an integer value for the arc center y-coordinate. */

aggregate_item = DDIF$_ARC_CENTER_Y;
integer_value = 150;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_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$_ARC_RADIUS_X_C;
integer_value = DDIF$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 );

/* Store an integer value for the arc radius x value. */

aggregate_item = DDIF$_ARC_RADIUS_X;
integer_value = 2000;
status = cda$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)

```
/* Store "ANGLE CONSTANT" as the data type choice for the arc
   start value. */

aggregate_item = DDIF$_ARC_START_C;
integer_value = DDIF$K_ANGLE_CONSTANT;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_length,
                         &integer_value );
if( FAILURE( status ) )
    return ( status );

/* Store an integer value for the arc start value. */

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$_ARC_EXTENT_C;
integer_value = DDIF$K_ANGLE_CONSTANT;
status = cda$store_item( &root_aggregate_handle,
                         &aggregate_handle_stack[ahs_index],
                         &aggregate_item,
                         &integer_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_hrd_dir ( &root_aggregate_handle,      71
                          &previous_aggregate_handle,
                          &aggregate_handle_stack[ahs_index],
                           DDIF$K_DIR_NEW_LINE );
if( FAILURE( status ) )
    return ( status );
```

(continued on next page)

```
    previous_aggregate_handle = aggregate_handle_stack[ahs_index];
    status = create_hrd_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,      (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)**

```
/*
** 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   integer_length = sizeof( integer_value );
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$_HRD_DIRECTIVE;
    integer_value = dir_type;
    status = cda$store_item(root_handle,
                            &aggregate_handle,
                            &aggregate_item,
                            &integer_length,
                            &integer_value );
    if( FAILURE( status ) )
        return ( status );

    *return_handle = aggregate_handle;

    return(1 );
}
```

**(continued on next page)**

```
/*
** 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;
```

(continued on next page)

**Example 10–1 (Cont.): Sample CDA Toolkit Program**

```
    return(1 );
}
```

Example 10–2 illustrates the Analysis output of the DDIF document created by Example 10–1. The callouts in Example 10–1 correspond to the callouts in Example 10–2.

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 10–1; instead, the default values specified by the CDA Toolkit are accepted.

**Example 10–2: Analysis Output of DDIF File**

```
              DDIF_DOCUMENT
1 {
3  DDF_DESCRIPTOR
2  {
4   DSC_MAJOR_VERSION  1  ! Longword Integer
                DSC_MINOR_VERSION  0  ! Longword Integer
                DSC_PRODUCT_IDENTIFIER  "DDIF$"
                DSC_PRODUCT_NAME
                (
5    ISO_LATIN1  "Test V1.0"
                )
               }
7  DDF_HEADER
6  {
9   DHD_EXTERNAL_REFERENCES
8   {
10  ERF_DATA_TYPE
                  (
```

(continued on next page)

```
                1  ! Object Identifier
                3  ! Object Identifier
                12  ! Object Identifier
                1011  ! Object Identifier
                1  ! Object Identifier
                3  ! Object Identifier
                1  ! Object Identifier
               )
               ERF_DESCRIPTOR
               (
(11)    ISO_LATIN1  "Style Guide"
               )
(12)   ERF_LABEL  ISO_LATIN1  "defstyle"  ! Char. string.
               ERF_LABEL_TYPE  STYLE_LABEL_TYPE  "$STYLE"
               ERF_CONTROL  NO_COPY_REFERENCE  ! Integer = 2
              }

              DHD_LANGUAGES_C
              (
(13)   ISO_639_LANGUAGE  ! Integer = 0
(15)   ISO_639_LANGUAGE  ! Integer = 0
              )
              DHD_LANGUAGES
              (
(14)   "E/USA/"
(16)   "Mandarin"
              )
              DHD_STYLE_GUIDE  1  ! Longword Integer
             }
(18) DDF_CONTENT
(17) {
(20)  SEG_CONTENT
(19)  {
(21)   SEG_SEGMENT_TYPE  "PARA"
(23)   SEG_SPECIFIC_ATTRIBUTES
(22)   {
(25)    SGA_TYPE_DEFNS
(24)    {
(26)     TYD_LABEL  "FRAME"
(28)     TYD_ATTRIBUTES
(27)     {
(29)      SGA_CONTENT_CATEGORY  TWOD_CATEGORY  "$2D"
(30)      SGA_FRM_FLAGS  "%B01000000000000000000000000000000"  ! Flags
(31)      SGA_FRM_BOX_LL_X_C  VALUE_CONSTANT  ! Integer = 0
(32)      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]
                SGA_FRMGLY_HORIZONTAL  FMT_CENTER_OF_PATH  ! Integer = 2  [Default]
               }
              }
             }
```

(continued on next page)

```
34  SEG_CONTENT
33  {
35   TXT_CONTENT  "This is the first line of the example text."
                 }
36  {
                  HRD_DIRECTIVE  DIR_SPACE  ! Integer = 5
                 }
37  {
                  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.
                 }
38  {
                  HRD_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
                 }
                 {
                  GTX_CONTENT  ISO_LATIN1  "The third line of the example text will
 begin on a new line."  ! Char. string.
                 }

                 {
                  HRD_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
                 }
                 {
                  HRD_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
                 }
                 {
                  GTX_CONTENT  ISO_LATIN1  "The fourth line of the example text will be
 separated from the previous lines by a blank line, and will be the last
 text on the first page."  ! Char. string.
                 }
                 {
                  HRD_DIRECTIVE  DIR_NEW_PAGE  ! Integer = 1
                 }
                 {

                  GTX_CONTENT  ISO_LATIN1  "The following is a Bezier curve, using the
 same path as the polyline, within a frame:"  ! Char. string.
                 }
39  {
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
                   )
```

(continued on next page)

```
                  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_LINE  ! Integer = 2
                }
                {
                 HRD_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
                }
49  {
                 GTX_CONTENT  ISO_LATIN1  "The following is a polyline
within a frame:"  ! Char. string.
                }
50  {
                 HRD_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
                }
                {
                 HRD_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
                }
                {
                 HRD_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
                }
                {
                 HRD_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
                  )
```

(continued on next page)

**Example 10–2 (Cont.): Analysis Output of DDIF File**

```
                    BEZ_PATH
                    (
(56)     500  ! Integer
(58)     500  ! Integer
                     2500  ! Integer
                     2000  ! Integer
                     3500  ! Integer
                     2000  ! Integer
                     5500  ! Integer
                     500  ! Integer
                    )
                   }
                  }

(59)  {
                   HRD_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
                  }
                  {
                   HRD_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
                  }
(60)  {
                   GTX_CONTENT  ISO_LATIN1  "The following is a
basketweave-filled arc within a frame:"  ! Char. string.
                  }
(61)  {
                   HRD_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
                  }
                  {
                   HRD_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
                     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)    }
```

(continued on next page)

```
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]
                        ARC_RADIUS_DELTA_Y  0  ! Longword Integer  [Default]
                        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]
                        ARC_ROTATION "%F0.000000e+00"  ! Single Prec. Floating Point [Default]
                       }
                      }
                     }

71     {
                      HRD_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
                     }
                     {
                      HRD_DIRECTIVE  DIR_NEW_LINE  ! Integer = 2
                     }
72     {
                      GTX_CONTENT  ISO_LATIN1  "This ends the examples."  ! Char. string.
                     }
                    }
                   }
                  }
```

Chapter 11

# CDA Converter Routines

This chapter provides detailed discussions of the converter routines and the VMS and ULTRIX compile and link procedures that applications must create in writing CDA-conforming front and back ends. Each routine description includes the following information:

- A routine definition that each application must name according to its operating system-specific format
- Descriptions of each routine argument
- A description of the routine itself
- A list of possible values returned by each routine argument

**NOTE**

The entry points and conventions defined throughout this reference section must be followed on both VMS and ULTRIX systems in order for all front and back ends to work properly with the CDA Converter Kernel.

If you are programming in Ada, please refer to the *Guide to Applications Programming* for information on Ada programming with DECwindows.

## 11.1 Compile and Link Procedures for Converter Images

To create a VMS or ULTRIX front or back end image using the CDA Toolkit routines, include the following public files in your source code:

| VMS and ULTRIX File Names | Description |
|---|---|
| SYS$LIBRARY:cda$def.h<br>/usr/include/cda_def.h | CDA Toolkit keyword definitions |
| SYS$LIBRARY:ddif$def.h<br>/usr/include/ddif_def.h | DDIF aggregate definitions |
| SYS$LIBRARY:dtif$def.h<br>/usr/include/dtif_def.h | DTIF aggregate definitions |
| SYS$LIBRARY:cda$msg.h<br>/usr/include/cda_msg.h | CDA error messages |

Section 11.1.1 describes the VMS compile and link procedure for CDA converters. Section 11.1.2 describes the ULTRIX compile and link procedure for CDA converters.

### 11.1.1 VMS Compile and Link Procedure

You can compile and link a front end on VMS using the following build procedure:

```
$ cc ddif$cvt_src:cda$rtx
$ link /share=ddif$read_text -
        sys$input:/option
identification="DECwindows v1.0"
universal = ddif$read_text
gsmatch=lequal,1,0
psect=$char_string_constant,shr,exe,nowrt
psect=$data,shr,exe,nowrt
cda$rtx
sys$library:cda$access/share
sys$share:vaxcrtl/share
```

You can compile and link a back end on VMS using the following build procedure:

```
$ cc ddif$cvt_src:cda$rtx
$ link /share=ddif$write_text -
        sys$input:/option
identification="DECwindows v1.0"
universal = ddif$write_text
gsmatch=lequal,1,0
psect=$char_string_constant,shr,exe,nowrt
psect=$data,shr,exe,nowrt
cda$rtx
sys$library:cda$access/share
sys$share:vaxcrtl/share
```

Note that this compile and link procedure does not provide debugging.

When the build is complete, define a logical named **domain**$read_**format** that points to your executable image so that the CONVERT/DOCUMENT or cdoc command will locate your converter when the CDA$_INPUT_FORMAT value (for front ends) or the CDA$_OUTPUT_FORMAT value (for back ends) is passed into the std-item list.

### 11.1.2 ULTRIX Compile and Link Procedure

You can compile and link a front end on ULTRIX using the following build procedure:

```
% cc -o ddif_read_myformat ddif_read_myformat.c -
     -lddif_fe -lddif -lm
```

You can compile and link a back end on ULTRIX using the following build procedure:

```
% cc -o ddif_write_myformat ddif_write_myformat.c -
     -lddif_be -lddif -lm
```

In the build procedure for a front end or back end on ULTRIX systems, the **-lm** parameter specifies the math library that is required by the CDA Toolkit routines (**-lddif**).

When the build procedure on ULTRIX systems is complete, you should store the output file in the /usr/bin directory for use with the **cdoc** command. Alternatively, you can define CDAPATH (an environment variable) to be the directory containing your front end or back end. The CDA Converter Kernel searches CDAPATH first when you invoke the **cdoc** command.

# *Close* Entry Point

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 front end. The *close* routine is one of the routines that compose a front end.

## FORMAT

**status = close-procedure**

*(front-end-context)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | Usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **front-end-context** | Usage: | context |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

## RETURNS

***status***
A condition value indicating the return status of the routine call.

***front-end-context***
Context returned by either ddif$read_**format** or dtif$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.

## *Close* Entry Point

### Description

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

In the aggregate method of 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, DTIF, 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.

The name of this routine is defined by the user. The front end simply returns the address of this routine to the CDA Converter Kernel.

### RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

The *close* routine can also return any front end-specific error conditions.

# *Get-Aggregate* Entry Point

Returns the next aggregate in the document to the converter kernel. The *get-aggregate* routine is one of the routines that compose a front end.

## FORMAT

**status = get-aggregate-procedure**

*(front-end-context ,aggregate-handle ,aggregate-type)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | Usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **front-end-context** | Usage: | context |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **aggregate-handle** | Usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **aggregate-type** | Usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## RETURNS

***status***
A condition value indicating the return status of the routine call.

***front-end-context***
Context returned from either ddif$read_**format** or dtif$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***
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***
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.

## Description

Depending on the conversion used, the *get-aggregate* routine either creates and populates the next document content aggregate (aggregate method of conversion) or it reads the next aggregate from the in-memory document (document method of conversion). 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.

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.

Before creating any of the document content aggregates, the *get-aggregate* routine must first create the required aggregates. For document data, the required aggregates are DDIF$_DSC, DDIF$_DHD, and DDIF$_SEG. For table data, the required aggregates are DTIF$_HDR, DTIF$_DSC, and DTIF$_TBL. 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 11–1.

**Table 11–1: 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$_EOS | End of segment |
| DTIF$_DSC | Table descriptor |
| DTIF$_HDR | Table header |
| DTIF$_TBL | Table definition |
| DTIF$_ROW | Table row |
| DTIF$_CLD | Table cell |

The name of this routine is defined by the user. The front end simply returns the address of this routine to the CDA Converter Kernel.

## RETURN VALUES

| Return Value | Description |
|---|---|
| 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.

# *Get-Position* Entry Point

Returns the current position in and total size of the current data stream. The *get-position* routine is one of the routines that compose a front end.

## FORMAT

**status = get-position-procedure**

*(front-end-handle ,stream-position ,stream-size)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | Usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **front-end-handle** | Usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **stream-position** | Usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |
| **stream-size** | Usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by reference |

## RETURNS

***status***
A condition value indicating the return status of the routine call.

***front-end-handle***
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 either ddif$read_**format** or dtif$read_**format**.

***stream-position***
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***
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-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.

The name of this routine is defined by the user. The front end simply returns the address of this routine to the CDA Converter Kernel.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

# *domain*$read_*format* Entry Point

Initializes the conversion process and establishes any special processing information for the front end. The **domain**$read_**format** routine is one of the routines that compose a front end.

## FORMAT

**status = domain$read_format** *(standard-item-list ,converter-context ,front-end-context ,get-aggregate-procedure ,get-position-procedure ,close-procedure)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | Usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **standard-item-list** | Usage: | item_list_2 |
| | Data type: | record |
| | Access: | read only |
| | Mechanism: | by reference,array reference |
| **converter-context** | Usage: | context |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **front-end-context** | Usage: | context |
| | Data type: | longword |
| | Access: | write only |
| | Mechanism: | by reference |

| Argument | Argument Information | |
|---|---|---|
| **get-aggregate-procedure** | Usage: | procedure |
| | Data type: | procedure entry mask |
| | Access: | write only |
| | Mechanism: | by reference |
| **get-position-procedure** | Usage: | procedure |
| | Data type: | procedure entry mask |
| | Access: | write only |
| | Mechanism: | by reference |
| **close-procedure** | Usage: | procedure |
| | Data type: | procedure entry mask |
| | Access: | write only |
| | Mechanism: | by reference |

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Arguments

***standard-item-list***
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:

| | | |
|---|---|---|
| item code | buffer length | 0 |
| buffer address | | 4 |

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. For more information on the calling sequence for a user *get* routine, see the description of the *Get* entry point.

- 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. For more information on the *get-position* procedure, see the description of the *Get-Position* entry point.

- 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.

**NOTE**

> If, in processing the standard item list, you encounter an unrecognized item, your front or back end should ignore that item and not return an error.

#### *converter-context*

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*

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***
Receives the address of the *get-aggregate* routine. The **get-aggregate-procedure** argument receives the address of this procedure entry mask. For more information on the calling sequence for the *get-aggregate* routine, see the description of the *Get-Aggregate* entry point.

***get-position-procedure***
Receives the address of the *get-position* routine. The **get-position-procedure** argument receives the address of this procedure entry mask. For more information on the calling sequence for the *get-position* routine, see the description of the *Get-Position* entry point.

***close-procedure***
Receives the address of the *close* routine. The **close-procedure** argument receives the address of this procedure entry mask. For more information on the calling sequence for the *close* routine, see the description of the *Close* entry point.

---

## Description

The read_**format** entry points (ddif$read_**format** and dtif$read_**format**) are the initial entry points in the front end. The read_**format** 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.

On ULTRIX systems, front end and back end converters are invoked as subprocesses, rather than being dynamically loaded into the same address space, as they are on VMS systems. Because of this, on ULTRIX systems the entry point name for the read_**format** entry point is always the same, cda$read_format, no matter what the format name. This is usually accomplished in source code by using a "jacket" routine named cda$read_format that simply calls the real routine (ddif$read_**format** or dtif$read_**format**) with the same parameters (see the Text front end source code example in Chapter 12). Another way of accomplishing this is to use compiler directives (#ifdef in the C language) to name the function differently, depending on the operating system.

In order to initialize a document or aggregate method of conversion, the ddif$read_**format** or dtif$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 for the **standard-item-list** argument.
- cda$w_item_address specifies the address of the item.

These fields are defined in the file cda$def.h on VMS systems and in the file cda_def.h on ULTRIX systems.

In addition, the ddif$read_**format** or dtif$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, DTIF, 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$read_**format** or dtif$read_**format** routine must call the CREATE ROOT AGGREGATE routine to create the document root aggregate. In the case of aggregate-method conversion, once the root aggregate is created, control passes back to the CDA Converter Kernel.

In the case of document-method conversion, the read_**format** routine must also create the appropriate aggregates before reading the entire document from the input stream and placing it in memory. For document data, the required aggregates are DDIF$_DSC, DDIF$_DHD, and DDIF$_SEG. For table data, the required aggregates are DTIF$_HDR, DTIF$_DSC, and DTIF$_TBL. Once the entire document is in memory, the ddif$read_**format** or dtif$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, DTIF, or Text, the 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 CDA Converter Kernel.

## RETURN VALUES

| Return Value | Description |
|---|---|
| CDA$_NORMAL | Normal successful completion |

The read_**format** entry point can also return any front end-specific error conditions.

# *domain*$write_*format* Entry Point

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 **domain**$write_**format** routine composes a back end.

## FORMAT

**status = domain$write_format** *(function-code ,standard-item-list ,private-item-list ,front-end-handle ,back-end-context)*

## Argument Information

| Argument | Argument Information | |
|---|---|---|
| **status** | Usage: | cond_value |
| | Data type: | longword (unsigned) |
| | Access: | write only |
| | Mechanism: | by value |
| **function-code** | Usage: | longword_unsigned |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |
| **standard-item-list** | Usage: | item_list_2 |
| | Data type: | record |
| | Access: | read only |
| | Mechanism: | by reference, array reference |
| **private-item-list** | Usage: | unspecified |
| | Data type: | unspecified |
| | Access: | read only |
| | Mechanism: | by reference |
| **front-end-handle** | Usage: | identifier |
| | Data type: | longword (unsigned) |
| | Access: | read only |
| | Mechanism: | by reference |

| Argument | Argument Information | |
|---|---|---|
| **back-end-context** | Usage: | context |
| | Data type: | longword (unsigned) |
| | Access: | read only or write only |
| | Mechanism: | by reference |

## RETURNS

***status***
A condition value indicating the return status of the routine call.

***function-code***
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.h on VMS systems and in the file cda_def.h on ULTRIX systems. 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 either ddif$write_**format** or dtif$write_**format** returned the value CDA$_SUSPEND. If CDA$_SUSPEND is returned by a call to the write_**format** routine, 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 the write_**format** routine returned the value CDA$_SUSPEND.

  If CDA$_SUSPEND is returned by a call to ddif$write_**format** or dtif$write_**format**, either CDA$_STOP or CDA$_CONTINUE must be specified so that resources locked by the conversion may be released.

***standard-item-list***
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:

| item code | buffer length | 0 |
| --- | --- | --- |
| buffer address | | 4 |

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. For more information on the calling sequence for a user *put* routine, see the description of the *Put* routine.

- 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.

**NOTE**

> If, in processing the standard item list, you encounter an unrecognized item, your front or back end should ignore that item and not return an error.

***private-item-list***
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.

On ULTRIX systems, the CDA$_OUTPUT_BACK_END_PROCEDURE item must be specified at the CONVERT routine call for this parameter to be used.

***front-end-handle***
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***
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** or dtif$write_**format** for the functions CDA$_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.

## Description

The write_**format** entry points (ddif$write_**format** and dtif$write_**format**) are the entry points 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.

On ULTRIX sytems, front end and back end converters are invoked as subprocesses, rather than being dynamically loaded into the same address space, as they are on VMS systems. Because of this, on ULTRIX systems the entry point name for the write_**format** entry point is always the same, cda$write_format, no matter what the format name. This is usually accomplished in source code by using a "jacket" routine named cda$write_format that simply calls the real routine (ddif$write_**format** or dtif$write_**format**) with the same parameters (see the Text front end source code example in Chapter 12). Another way of accomplishing this is to use compiler directives (#ifdef in the C language) to name the function differently depending on the operating system.

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, DTIF, 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 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.h on VMS systems and in the file cda_def.h on ULTRIX systems.

## RETURN VALUES

| Return Value | Description |
|---|---|
| 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. |

The write_**format** routine can also return any error returned by the specific front end or the specific back end.

Chapter 12

# Text Front End Source File

This chapter 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 chapter, 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.

1. All of these routines from the CDA Toolkit are used by the Text front end.
2. These are the additional entry points in the Text front end.
3. 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
**
**/

#ifdef vms
#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 */


#else
#include <ddif_def.h>
#include <cda_def.h>
#include <cda_msg.h>
#endif

#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 previous 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$READ_FORMAT().
**      This routine is the jacket entry point for the text Front End on
**      Ultrix.  It is called from the converter kernel to
**      call the "real" entry point which initializes the conversion.
**      When employed on VMS systems, this routine is not called (or even
**      compiled). On VMS systems, the converter kernel calls the
**      DDIF$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
**
**--
**/

#ifdef ultrix
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 aggregate (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, 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.

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.

⓫ 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:                                             ⓫
**
**      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 ❿.

```
/*
**++
**  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 an 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 ❿ 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 ❿. Because the state is still set to DDIF$_SEG, ❿ 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 ❿.

```
/*
**++                                                                    ⓭
**  FUNCTIONAL DESCRIPTION:
**
**      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 ❿.

⓳ This loop reads each character on the line of text. If a form-feed character is encountered, 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:                                          ⑯
**
**      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)            ⓱
                {
                        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)   19
            {
            /* 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)                     20
                     {
                             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.

❸❻ 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; }
```

# Chapter 13

# CDA Viewer Routines

This chapter describes the VMS and ULTRIX compile and link procedures and routines used to write a viewer application.

There are two sets of viewer routines: 1) the character cell viewer routines, which are listed first and which are preceded by DvrCC, and 2) the DECwindows viewer routines. Each routine description includes the following information:

- An ULTRIX C style binding that is supported on both VMS and ULTRIX systems
- A description of the value returned by the routine
- A description of each routine argument
- A description of the routine itself
- A list of possible values returned by the routine

## 13.1 CDA Viewer Support of Adobe Font Metrics

The CDA Viewer uses the Adobe font metrics in processing a DDIF file for viewing. The font name in a DDIF file follows the X-11 font naming convention. When processing a file from a creating application that uses font metrics other than Adobe font metrics, the CDA Viewer defaults to the Adobe Courier font.

The DECwindows CDA Viewer queries the X server for a list of available fonts when processing a file for viewing. Although the CDA Viewer does not use these fonts in its calculations, it tries to match the font from the file with an X11 font on the server. If there is not an exact match, the CDA Viewer uses the font from the list that is the closest lower point size for that font name. If there is no match at all, the DECwindows CDA Viewer display type defaults to 12 point Adobe Courier.

The character cell CDA Viewer displays all files in a 12 point Courier font. The contents of each file are spaced and displayed correctly, based on the font that is stored in the file.

The Adobe font metrics are stored in SYS$PS_FONT_METRICS:.AFM on VMS systems and in /usr/lib/font/metrics/ on ULTRIX systems.

## 13.2 Compile and Link Procedures for Viewer Images

To create a VMS or ULTRIX program using the CDA Viewer callable interface, include the following public files in your source code:

| VMS and ULTRIX File Names | Description |
|---|---|
| SYS$LIBRARY:DVR$MSG.H<br>/usr/include/dvr_msg.h | Status codes for both the character cell viewer and the DECwindows viewer callable interfaces. |
| SYS$LIBRARY:DVR$CC_DEF.H<br>/usr/include/dvr_cc_def.h | Literals and structure definitions for the character cell viewer callable interface. |
| SYS$LIBRARY:DVR$DECW_DEF.H<br>/usr/include/X11/dvr_decw_def.h | Literals and structure definitions for the DECwindows viewer callable interface. |

On ULTRIX systems, you must also install the DECimage Application Services libraries (libimg.a, libids.a, and libchf.a) before you can use the CDA DECwindows viewer callable interface library (libdvr.a) and the CDA character cell viewer callable interface library (libdvs.a).

Section 13.2.1 describes the VMS compile and link procedure for CDA viewers. Section 13.2.2 describes the ULTRIX compile and link procedure for CDA viewers.

### 13.2.1 VMS Link Procedure

After you compile your source code into an object module (for example, **YOUR_MODULE**.OBJ), link a C program (VIEWER_PROGRAM.EXE) using the following link command on VMS:

```
$ LINK/EXE=VIEWER_PROGRAM YOUR_MODULE.OBJ, -
        SYS$INPUT/OPT
        SYS$SHARE:DDIF$VIEWSHR/SHARE
```

### 13.2.2 ULTRIX Link Procedures

After you compile your source code into an object module (for example, **your_module**.o), link a DECwindows viewer program (dw_viewer_program) using the following link command:

```
csh> cc -o dw_viewer_program    \
           your_module.o        \
           /usr/lib/libdvr.a    \
           /usr/lib/libids.a    \
           /usr/lib/libdwt.a    \
           /usr/lib/libimg.a    \
           /usr/lib/libchf.a    \
           /usr/lib/libX11.a    \
           /usr/lib/libdvs.a    \
           /usr/lib/libddif.a   \
           /usr/lib/libm.a
```

To link a character cell viewer program (cc_viewer_program), use the following command:

```
csh> cc -o cc_viewer_program      \
           your_module.o          \
           /usr/lib/libdvs.a      \
           /usr/lib/libimg.a      \
           /usr/lib/libchf.a      \
           /usr/lib/libddif.a     \
           /usr/lib/libcurses.a   \
           /usr/lib/libtermlib.a  \
           /usr/lib/libm.a
```

Applications that call the viewer routines should use a general condition handling routine for asynchronous signals that the viewer may generate. The signals probably will occur when the viewer is processing images, rather than text or graphics. The following example is a condition handling routine shell, written in C, which can be included in applications that call the viewer:

```
int my_condition_handler(signal, mechanism)
 struct chf$signal_array              *signal;
 struct chf$mech_array                *mechanism;
{

  /* signal->chf$l_sig_name contains the error status;
   * process status and continue program execution
   */

}
```

In the main routine of your application, add the following call to set up the condition handler:

```
#ifdef VMS
       LIB$ESTABLISH (my_condition_handler);
#endif

#ifdef ultrix
        ChfEstablish (my_condition_handler);
#endif
```

LIB$ESTABLISH( ) is a VMS run-time library routine. ChfEstablish( ) is the condition handling establish routine provided within libchf.a.

# CC DELETE PAGE

Deallocates the page display structure allocated by the routine CC GET PAGE.

## C FORMAT

**status = DvrCCDeletePage** *(cc_viewer_context, line_array)*

## Argument Information

```
unsigned long   *cc_viewer_context;
unsigned char   ***line_array;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## ARGUMENTS

***cc_viewer_context***
The address of an unsigned longword that specifies the CC viewer context. This value must be the value returned by the CC INITIALIZE routine.

***line_array***
The address of an unsigned longword that contains the address of the line array returned by the CC GET PAGE routine. This parameter serves to identify the line array and the corresponding line size array to be deallocated.

## Description

The CC DELETE PAGE routine deallocates the page display structure allocated by the routine CC GET PAGE. Applications may delete this structure once it is no longer required. Page structures must be deallocated using the CC DELETE PAGE routine; applications cannot directly deallocate these structures.

## RETURN VALUES

| Return Value | Description |
|---|---|
| DVR$_NORMAL | Page successfully deleted |
| DVR$_MEMDEALLOFAIL | Memory deallocation failure |
| CDA$_xxxx | Any CDA return status |

# CC END

Deallocates all internal structures that were allocated and does general cleanup required for CC viewer shutdown for the current file.

## C FORMAT

**status = DvrCCEnd** *(cc_viewer_context)*

## Argument Information

```
unsigned long   *cc_viewer_context;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## ARGUMENTS

***cc_viewer_context***
The address of an unsigned longword that specifies the CC viewer context. This value must be the value returned by the CC INITIALIZE routine.

## Description

The CC END routine deallocates all internal structures that were allocated and does general cleanup required for CC viewer shutdown for the current file. This routine may be called at any point during document processing. Any outstanding page structures not previously deleted by calls to the CC DELETE PAGE routine are also deallocated.

## RETURN VALUES

| Return Value | Description |
|---|---|
| DVR$_NORMAL | Structures successfully deallocated |
| DVR$_MEMDEALLOFAIL | Memory deallocation failure |
| CDA$_xxx | Any CDA return status |

# CC GET PAGE

Returns the next sequential formatted page from the CDA document.

## C FORMAT

**status = DvrCCGetPage**

*(cc_viewer_context, number_of_lines, line_array, line_size_array)*

## Argument Information

```
unsigned long       *cc_viewer_context;
unsigned long       *number_of_lines;
unsigned char       ***line_array;
unsigned long       **line_size_array;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## ARGUMENTS

***cc_viewer_context***
The address of an unsigned longword that specifies the CC viewer context. This value must be the value returned by the CC INITIALIZE routine.

***number_of_lines***
The address of an unsigned longword that receives the number of lines in this page.

***line_array***
The address of an unsigned longword that receives the address of an array of longwords in which each element is the address of a null-terminated character string that represents the characters to be displayed on a line. Each element in the array represents a specific line number. Element 0 represents line 1, element 1 represents line 2, and so on.

***line_size_array***
The address of an unsigned longword that receives the address of an array of longwords in which each element is the length of the character string for the corresponding line-array element. If you specify 0 by value for this parameter, no size array is returned.

## Description

The CC GET PAGE routine returns the next sequential formatted page from the CDA document. The page is returned as an array of character string pointers. Each character string represents a line of text. After the last page in the document has been processed, the CC GET PAGE routine returns a null page structure and the status DVR$_EOD (end of document). The page structures remain in memory until they are explicitly deleted by a call to either the CC DELETE PAGE routine or the CC END routine.

## RETURN VALUES

| Return Value | Description |
|---|---|
| DVR$_NORMAL | Page returned successfully. |
| DVR$_EOD | The application is at the bottom of the file and cannot page forward any further. |

Any other error status codes.

# CC INITIALIZE

Initializes the character-cell CDA Viewer and returns a context block to the caller for use in subsequent character-cell CDA Viewer routine calls.

## C FORMAT

**status = DvrCCInitialize**

*(select_options, standard_item_list, private_item_list, display_height, display_width, cc_viewer_context)*

## Argument Information

```
unsigned long       select_options;
ITEM_LIST_TYPE     *standard_item_list;
ITEM_LIST_TYPE     *private_item_list;
unsigned long       display_height;
unsigned long       display_width;
unsigned long      *cc_viewer_context;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## ARGUMENTS

***select_options***
A flag vector that may contain any of the following CDA Viewer masks:

| Mask Value | Meaning |
|---|---|
| DVR$M_Outfile | Output is directed to a text file. If set, the entire document is written at once to the output file (or standard output) in the CC INITIALIZE routine without the application having to call the CC GET PAGE routine or the CC END routine. |
| DVR$M_SoftDirectives | Obey DDIF soft directives (new line, new page, and so on). |

| Mask Value | Meaning |
|---|---|
| DVR$M_Auto_Wrap | Output is word wrapped at the specified page width or galley width. |
| DVR$M_Text | Set to create text output. |
| DVR$M_Graphics | Set to note location of graphics in the output with a replacement message. |
| DVR$M_Images | Set to note location of images in the output with a replacement message. |
| DVR$M_Layout | Use generic layout. |
| DVR$M_SpecificLayout | Use generic and specific layout. |
| DVR$M_ReportErrors | Write all nonfatal error messages to SYS$ERROR or **stderr**. Fatal errors are always reported. |
| DVR$M_Paging | If not set, the entire document is written at once to the output file (or standard output) in the CC INITIALIZE routine without the application having to call the CC GET PAGE routine or the CC END routine. |
| DVR$M_Text_Backend | If set, the CC viewer acts as a text back end. It expects the CDA front end handle to be passed in the private item list, with item code DVR$_FRONT_END_HANDLE. |

These masks are defined in DVR$CC_DEF.H. Note that if DVR$M_Text is not set, there will be no text output.

***standard_item_list***
Address of a standard CDA item list. An item list contains entries consisting of two longwords. The item list is terminated by a null entry.

The item codes are the same CDA$ item codes accepted by the CONVERT routine in its **standard_item_list** parameter. The CDA item codes are defined in SYS$LIBRARY:cda$def.* on VMS systems and in /usr/include/cda_def.h on ULTRIX systems. Item codes of the same names, but with the DVR$ prefix, are provided in SYS$LIBRARY:DVR$CC_DEF.H on VMS systems and in /usr/include /dvr_cc_def.h on ULTRIX systems.

The CDA$_INPUT_PROCEDURE and CDA$_INPUT_ PROCEDURE_ PARM codes are supported. These allow the calling application to supply DDIF input rather than having the CDA Viewer get it from the specified input file. The standard item list actually supports all the items listed for the CONVERT routine, although not all item combinations make sense.

***private_item_list***
The address of a private item list, in the same format as the **standard_item_ list**. This item list only supports DVR$ item codes. Currently, the only item codes expected in this private item list are shown in the following table.

| Item Code | Meaning |
|---|---|
| DVR$_FRONT_END_HANDLE | Front end input procedure handle |
| DVR$_PAGE_HEIGHT | Formatted page height in lines of characters |

| Item Code | Meaning |
|---|---|
| DVR$_PAGE_WIDTH | Formatted page width in columns of characters |

***display_height***
The maximum height per page (in rows). If you specify 0 for this parameter, the resulting screen height is set to a size adequate to include the entire original page. This is useful for applications that would like the entire page formatted to a specific number of rows.

***display_width***
The maximum page width (in columns). If you specify 0 for this parameter, the resulting screen width defaults to 132 columns.

***cc_viewer_context***
The address of an unsigned longword that receives the CC viewer context. The address of this value must be specified as the **cc_viewer_context** input parameter during calls to the other CC routines.

## Description

The CC INITIALIZE routine initializes the character-cell CDA Viewer and returns a context block to the caller for use in subsequent character-cell CDA Viewer routine calls.

## RETURN VALUES

| Return Value | Description |
|---|---|
| DVR$_NORMAL | CC viewer successfully initialized |

Any error status codes.

# BOTTOM DOCUMENT

Displays the last page of content in the file in the widget window.

## C FORMAT

**status = DvrBottomDocument** *(w)*

## Argument Information

```
int             status;
Widget          w;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## ARGUMENTS

***w***
Identifier of the CDA Viewer widget.

## Description

When an application calls the BOTTOM DOCUMENT routine, the CDA Viewer displays the last page of content.

# BOTTOM DOCUMENT

## RETURN VALUES

| Return Value | Description |
|---|---|
| DVR$_NORMAL | The last page of content was displayed successfully. |
| DVR$_EOD | The application is at the bottom of the file and cannot page forward any further. |
| DVR$_ERROR | An error was encountered while reading the file, or converting to in-memory DDIF. |
| DVR$_INVADDR | Invalid address. |
| DVR$_FILENOTOPEN | There is no open file. |
| CDA$_xxxx | Any CDA return status. |

# CLOSE FILE

Closes the file currently being read by the CDA Viewer and clears the window.

## C FORMAT

**status = DvrCloseFile** *(w)*

## Argument Information

```
int         status;
Widget      w;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## ARGUMENTS

***w***
The identifier of the CDA Viewer widget.

## Description

The CLOSE FILE routine closes the file currently being read by the CDA Viewer and clears the window.

## RETURN VALUES

| Return Value | Description |
|---|---|
| DVR$_NORMAL | The file was closed successfully. |
| DVR$_FILENOTOPEN | There is no open file. |
| DVR$_INVADDR | Invalid address. |
| CDA$_xxxx | Any CDA return status. |

# DOCUMENT INFO

Returns information from the header aggregate of the currently open document.

## C FORMAT

**status = DvrDocumentInfo** *(w, buffer_dsc)*

## Argument Information

```
int             status;
Widget          w;
char            **buffer_dsc;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## ARGUMENTS

***w***
The identifier of the CDA Viewer widget.

***buffer_dsc***
The address of a string buffer to be allocated.

## Description

The DOCUMENT INFO routine returns information from the header aggregate of the currently open document. This information includes the document title, author, version, and creation date.

## RETURN VALUES

| Return Value | Description |
|---|---|
| DVR$_NORMAL | The document's header was successfully read. |
| DVR$_BADPARAM | An invalid parameter was specified. |
| DVR$_FILENOTOPEN | There is no open file. |
| DVR$_DRMSTRINGFETCHFAIL | Failure to fetch a string. |
| DVR$_NODISPCONT | The requested information is not contained in the document. |
| DVR$_MEMDEALLOFAIL | Failure to deallocate memory. |
| DVR$_MEMALLOFAIL | Failure to allocate memory. |

# GOTO PAGE

Attempts to move to the specified page number.

## C FORMAT

**status = DvrGotoPage** *(w, page_num)*

## Argument Information

```
int        status;
Widget     w;
int        page_num;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## ARGUMENTS

***w***
Identifier of the CDA Viewer widget.

***page_num***
Page number of the desired page in the document.

## Description

The GOTO PAGE routine attempts to move to the specified page number.

## RETURN VALUES

| Return Value | Description |
|---|---|
| DVR$_NORMAL | The CDA Viewer widget has successfully moved to the requested page. |

| Return Value | Description |
|---|---|
| DVR$_EOD | End of document. |
| DVR$_PAGENOTFOUND | A page with the specified page number was not found in the document. |
| DVR$_BADPARAM | An invalid parameter was specified. |

# NEXT PAGE

Displays the next page of a CDA document.

## C FORMAT

**status = DvrNextPage** *(w)*

## Argument Information

```
int         status;
Widget      w;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## ARGUMENTS

***w***
Identifier of the CDA Viewer widget.

## Description

The NEXT PAGE routine displays the next page of a CDA document.

## RETURN VALUES

| Return Value | Description |
|---|---|
| DVR$_NORMAL | The CDA Viewer widget has successfully moved to the next page. |
| DVR$_INVADDR | Invalid address. |
| DVR$_EOD | End of document. |
| DVR$_FILENOTOPEN | There is no open file. |
| CDA$_xxxx | Any condition value returned by the CDA$ routines. |

# PREVIOUS PAGE

Displays the previous page (if one exists) of a CDA document.

## C FORMAT

**status = DvrPreviousPage** *(w)*

## Argument Information

```
int            status;
Widget         w;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## ARGUMENTS

***w***
Identifier of the CDA Viewer widget.

## Description

The PREVIOUS PAGE routine displays the previous page (if one exists) of a CDA document.

## RETURN VALUES

| Return Value | Description |
|---|---|
| DVR$_NORMAL | The CDA Viewer widget has successfully displayed the previous page. |
| DVR$_INVADDR | Invalid address. |

| Return Value | Description |
|---|---|
| DVR$_TOPOFDOC | The application is at the top of the file and cannot page backward any further. |
| DVR$_FILENOTOPEN | There is no open file. |
| CDA$_xxxx | Any condition value returned by the CDA$ routines. |

# REGISTER CLASS

Indicates that the CDA Viewer widget is registered with DRM.

## C FORMAT

**status = DvrRegisterClass** *( )*

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## Description

The REGISTER CLASS routine is used to indicate that the CDA Viewer widget is registered with DRM. This call is only necessary for developers using UIL.

## RETURN VALUES

| Return Value | Description |
|---|---|
| DVR$_NORMAL | The CDA Viewer widget was successfully registered with DRM. |
| DVR$_INVADDR | Invalid address. |
| DVR$_FAILURE | The CDA Viewer widget was not successfully registered with DRM. |

# TOP DOCUMENT

Displays the beginning content of the file in the widget window.

## C FORMAT

**status = DvrTopDocument** *(w)*

## Argument Information

```
int           status;
Widget        w;
```

## RETURNS

***status***
A condition value indicating the return status of the routine call.

## ARGUMENTS

***w***
The identifier of the CDA Viewer widget that opens and displays the information content of the file.

## Description

The TOP DOCUMENT routine displays the beginning content of the file in the widget window.

## RETURN VALUES

| Return Value | Description |
|---|---|
| DVR$_NORMAL | The beginning content was displayed successfully. |
| DVR$_TOPOFDOC | The application is at the top of the file and cannot page backward any further. |

| Return Value | Description |
|---|---|
| DVR$_INVADDR | Invalid address. |
| DVR$_FILENOTOPEN | There is no open file. |
| CDA$_xxxx | Any CDA return status. |

# VIEWER

Creates a widget for viewing a CDA file.

## C FORMAT

**Widget = DvrViewer**

*(parent, name, x, y, width, height, horz_scroll_bar, vert_scroll_bar, proc_options, callback, help_callback)*

## Argument Information

```
Widget         parent;
char           *name;
int            x;
int            y;
int            width;
int            height;
Boolean        horz_scroll_bar;
Boolean        vert_scroll_bar;
int            proc_options;
DwtCallbackPtr callback;
DwtCallbackPtr help_callback;
```

## RETURNS

***Widget***
Identifier of the created CDA Viewer widget.

## ARGUMENTS

***parent***
The parent window of the widget.

***name***
The name of the widget to be created.

***x***
A signed longword that defines in pixels the placement of the left side of the widget window relative to the inner upper left corner of the parent window. The default is 0.

***y***
A signed longword that defines in pixels the placement of the left side of the widget window relative to the inner upper left corner of the parent window. The default is 0.

***width***
The width in pixels of the widget window.

***height***
The height in pixels of the widget window.

***horz_scroll_bar***
A Boolean value indicating that a horizontal scroll bar should be included in the CDA Viewer widget.

***vert_scroll_bar***
A Boolean value indicating that a vertical scroll bar should be included in the CDA Viewer widget.

***proc_options***
An integer mask indicating the options for processing the document. For a list of possible processing options masks, see the low-level creation routine VIEWER CREATE.

***callback***
The identifier of the application routine to be called back. The callback routine should have the form **callback(Widget, tag, reason).**

***help_callback***
The identifier of the application help routine to be called back. The callback routine should have the form **callback(Widget, tag, reason).**

## CALLBACK ROUTINES

The format of the callback routines is as follows:

```
void  CallbackProc(WidgetID, tag, reason)
      Widget            *WidgetID;
      caddr_t           tag;
      DvrCallbackStruct *cb_data;
```

## CALLBACK DATA STRUCTURE

The format of the callback data structure is as follows:

```
typedef struct
{ int           reason;
  Xevent        *event;
  unsigned long status;
  char          *string_ptr;
} DvrCallbackStruct;
```

## CALL BACK REASONS

The application callback is called with the following values for **reason**:

***DvrCRactivated***
The CDA Viewer requests focus by clicking on MB1.

***DvrCRendDocument***
The end of the document has been displayed.

***DvrCRcdaError***
A nonrecoverable error was incurred while processing the document. See the status field of the callback structure for the specific status returned. See the string-ptr field for a character string describing the status.

***DvrCRhelpRequested***
Help was requested by clicking on HELP + MB1.

## CALLBACK FIELD DESCRIPTIONS

The callback field descriptions are as follows:

***reason***
See the Callback Reasons section.

***event***
A pointer to the X event structure describing the event that generated this callback.

***status***
The specific status returned.

***string_ptr***
The character string describing the status.

## Description

The VIEWER routine creates a widget that can be used to view the information content of an in-memory CDA document. If the document to be viewed is not in DDIF format, then a front end converter must exist that can convert the document from its non-DDIF format to in-memory DDIF. (A CDA Viewer widget can also be created using the low-level creation routine VIEWER CREATE.)

To associate a file with the CDA Viewer widget, see the description of the VIEWER FILE routine.

## RETURN VALUES

If the return value is successful, the VIEWER routine returns the ID of the widget. If the return value is failure, the VIEWER routine returns 0.

# VIEWER CREATE

Creates a widget for viewing a CDA file.

## C FORMAT

**Widget = DvrViewerCreate**

*(parent, name, override_arglist, override_argcount)*

## Argument Information

```
Widget   parent;
char     *name;
ArgList  override_arglist;
int      override_argcount;
```

## RETURNS

***Widget***
Identifier of the created CDA Viewer widget.

## ARGUMENTS

***parent***
The parent window of the widget.

***name***
The name of the widget to be created.

***override_arglist***
The application override argument list. This list consists of name/value pairs that describe the attributes of the created widget. For more information on the **override_arglist** argument, see the *VMS DECwindows Toolkit Routines Reference Manual.*

The **override_arglist** argument can contain any of the common arguments for low-level widget creation routines, plus the following widget-specific arguments:

- **Boolean horz_scroll_bar**
- **Boolean vert_scroll_bar**
- **int processing_options**
- **int paper_width**

- **int paper_height**

***override_argcount***
The number of arguments in the application override argument list. If there are no arguments in the argument list, then **override_argcount** must equal zero, but the **override_arglist** does not have to be null. In the *VMS DECwindows Toolkit Routines Reference Manual,* see the Common Arguments section for descriptions of the VAX and C formats of arguments common to all widgets.

## Widget Specific Attributes

***horz_scroll_bar***
Boolean argument that, if true, results in a horizontal scroll bar being placed at the bottom of the CDA Viewer widget window. The default value is true. The name of this argument is **DvrNscrollHorizontal**. This is a create time only resource and cannot be modified through XtSet Values.

***vert_scroll_bar***
Boolean argument that, if true, results in a vertical scroll bar being placed at the right side of the CDA Viewer widget window. The default value is true. The name of this argument is **DvrNscrollVertical**. This is a create time only resource and cannot be modified through XtSet Values.

***processing_options***
This argument is a mask formed by the union of processing options to be followed when document content is displayed in the CDA Viewer. The name of this argument is **DvrNprocessingOptions**. The supported processing options are shown in the following table.

| Processing Option | Mask Value |
|---|---|
| Word wrap | DvrWordWrap |
| Soft directives | DvrSoftDirectives |
| Layout | DvrLayout |
| Specific layout | DvrSpecificLayout |

The default is:

DvrWordWrap | DvrSoftDirectives | DvrLayout | DvrSpecificLayout

***paper_width***
Integer value that specifies the desired width of the paper in millimeters. The name of this argument is **DvrNpaperWidth**.

This argument does not apply to the CDA Viewer's window width; it applies only to the size of the paper to be used for each page displayed in the CDA Viewer's window. By default, the CDA Viewer uses the paper width stored in the document. If this item is not encoded, the CDA Viewer uses a default paper width of 8.5 inches.

***paper_height***
Integer value that specifies the desired height of the paper in millimeters. The name of this argument is **DvrNpaperHeight**.

This argument does not apply to the CDA Viewer's window height; it applies only to the size of the paper to be used for each page displayed in the CDA Viewer's window. By default, the CDA Viewer uses the paper height stored in the document. If this item is not encoded, the CDA Viewer uses a default paper height of 11 inches.

## Description

The VIEWER CREATE routine is a low-level routine used to create a widget for viewing an in-memory DDIF file. If the file is not in DDIF, there must be a front end converter that can be called to convert the file into in-memory DDIF. To associate a file with the CDA Viewer widget, see the description of the VIEWER FILE routine. A CDA Viewer widget can also be created using the high-level VIEWER creation routine.

See the description of the high-level creation routine VIEWER for the definition of the callback structure and the reasons for a callback.

Defaults:

The default height is 723 pixels and the default width is 684 pixels.

The default *x,y* location of the widget is the upper left corner of the parent.

A CDA Viewer widget can also be created using the high-level VIEWER routine.

## RETURN VALUES

If the return value is successful, the VIEWER CREATE routine returns the ID of the widget. If the return value is failure, the VIEWER CREATE routine returns 0.

# VIEWER FILE

Opens a file and begins to view the information content of the file, provided the file can be converted to in-memory DDIF.

## C FORMAT

**status = DvrViewerFile**

*(w, filename, format, optionsfile, getrtn, getprm)*

## Argument Information

```
int         status;
Widget      w;
char        *filename;
char        *format;
char        *optionsfile;
int         getrtn;
int         getprm
```

## RETURNS

***status***
The result of attempting to open and begin the conversion of the file.

## ARGUMENTS

***w***
The identifier of the CDA Viewer widget that should open and display the information content of the file.

***filename***
A character string specifying the name of the file to be viewed.

***format***
A character string specifying the format of the file to be viewed. If the file is not in DDIF format, a converter must exist that can convert the file to in-memory DDIF.

***optionsfile***
A character string specifying a file with processing options for the front end converter. If this argument is null, 0, or a zero-length string, it is not used.

***getrtn***
If the file should be read by other than the system default read routine, this argument should identify the alternative read routine. If the standard read routine is to be used, the value of **getrtn** should be 0.

***getprm***
The address of a longword used by **getrtn**. If the standard read routine is to be used, this value should be 0.

## Description

The VIEWER FILE routine opens a file and begins to convert the contents of the file to in-memory DDIF. If there is currently a file that is open, it is immediately closed, the viewer window is cleared, and the new file is opened.

## RETURN VALUES

| Return Value | Description |
|---|---|
| DVR$_NORMAL | The file was opened and a window of content was converted to in-memory DDIF and displayed in the CDA Viewer widget window. |
| DVR$_NOCONVERTER | There was no converter for the specified format. |
| DVR$_DDIFERROR | There was an error when converting the file to in-memory DDIF. |
| DVR$_FILENOTOPEN | The file could not be opened. |
| CDA$_xxxx | Any CDA return status. |

# Appendix A

# 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 A–1 describes each predefined fill pattern and shows its symbolic name and its corresponding DDIF pattern number. Figure A–1 illustrates each predefined fill pattern.

**Table A–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$K_PATT_VERT1_1 | 3 | The thickness ratio of black to white vertical lines in the pattern is 1 : 1. |
| DDIF$K_PATT_VERT1_3 | 4 | The thickness ratio of black to white vertical lines in the pattern is 1 : 3. |
| DDIF$K_PATT_VERT2_2 | 5 | The thickness ratio of black to white vertical lines in the pattern is 2 : 2. |

(continued on next page)

**Table A–1 (Cont.): DDIF Fill Patterns**

| Pattern Name | Number | Description |
|---|---|---|
| DDIF$K_PATT_VERT3_1 | 6 | The thickness ratio of black to white vertical lines in the pattern is 3 : 1. |
| DDIF$K_PATT_VERT1_7 | 7 | The thickness ratio of black to white vertical lines in the pattern is 1 : 7. |
| DDIF$K_PATT_VERT2_6 | 8 | The thickness ratio of black to white vertical lines in the pattern is 2 : 6. |
| DDIF$K_PATT_VERT4_4 | 9 | The thickness ratio of black to white vertical lines in the pattern is 4 : 4. |
| DDIF$K_PATT_VERT6_2 | 10 | The thickness ratio of black to white vertical lines in the pattern is 6 : 2. |
| DDIF$K_PATT_HORIZ1_1 | 11 | The thickness ratio of black to white horizontal lines in the pattern is 1 : 1. |
| DDIF$K_PATT_HORIZ1_3 | 12 | The thickness ratio of black to white horizontal lines in the pattern is 1 : 3. |
| DDIF$K_PATT_HORIZ2_2 | 13 | The thickness ratio of black to white horizontal lines in the pattern is 2 : 2. |
| DDIF$K_PATT_HORIZ3_1 | 14 | The thickness ratio of black to white horizontal lines in the pattern is 3 : 1. |
| DDIF$K_PATT_HORIZ1_7 | 15 | The thickness ratio of black to white horizontal lines in the pattern is 1 : 7. |
| DDIF$K_PATT_HORIZ2_6 | 16 | The thickness ratio of black to white horizontal lines in the pattern is 2 : 6. |
| DDIF$K_PATT_HORIZ4_4 | 17 | The thickness ratio of black to white horizontal lines in the pattern is 4 : 4. |
| DDIF$K_PATT_HORIZ6_2 | 18 | The thickness ratio of black to white horizontal lines in the pattern is 6 : 2. |
| DDIF$K_PATT_GRID4 | 19 | Each grid box has 4 units to a side. |
| DDIF$K_PATT_GRID8 | 20 | Each grid box has 8 units to a side. |
| DDIF$K_PATT_UPDIAG1_3 | 21 | The thickness ratio of black to white upward diagonal lines (going up from left to right) in the pattern is 1 : 3. |

(continued on next page)

**Table A–1 (Cont.): DDIF Fill Patterns**

| Pattern Name | Number | Description |
|---|---|---|
| DDIF$K_PATT_UPDIAG2_2 | 22 | The thickness ratio of black to white upward diagonal lines (going up from left to right) in the pattern is 2 : 2. |
| DDIF$K_PATT_UPDIAG3_1 | 23 | The thickness ratio of black to white upward diagonal lines (going up from left to right) in the pattern is 3 : 1. |
| DDIF$K_PATT_UPDIAG1_7 | 24 | The thickness ratio of black to white upward diagonal lines (going up from left to right) in the pattern is 1 : 7. |
| DDIF$K_PATT_UPDIAG2_6 | 25 | The thickness ratio of black to white upward diagonal lines (going up from left to right) in the pattern is 2 : 6. |
| DDIF$K_PATT_UPDIAG4_4 | 26 | The thickness ratio of black to white upward diagonal lines (going up from left to right) in the pattern is 4 : 4. |
| DDIF$K_PATT_UPDIAG6_2 | 27 | The thickness ratio of black to white upward diagonal lines (going up from left to right) in the pattern is 6 : 2. |
| DDIF$K_PATT_DOWNDIAG1_3 | 28 | The thickness ratio of black to white downward diagonal lines (going down from left to right) in the pattern is 1 : 3. |
| DDIF$K_PATT_DOWNDIAG2_2 | 29 | The thickness ratio of black to white downward diagonal lines (going down from left to right) in the pattern is 2 : 2. |
| DDIF$K_PATT_DOWNDIAG3_1 | 30 | The thickness ratio of black to white downward diagonal lines (going down from left to right) in the pattern is 3 : 1. |
| DDIF$K_PATT_DOWNDIAG1_7 | 31 | The thickness ratio of black to white downward diagonal lines (going down from left to right) in the pattern is 1 : 7. |
| DDIF$K_PATT_DOWNDIAG2_6 | 32 | The thickness ratio of black to white downward diagonal lines (going down from left to right) in the pattern is 2 : 6. |
| DDIF$K_PATT_DOWNDIAG4_4 | 33 | The thickness ratio of black to white downward diagonal lines (going down from left to right) in the pattern is 4 : 4. |

(continued on next page)

**Table A–1 (Cont.): DDIF Fill Patterns**

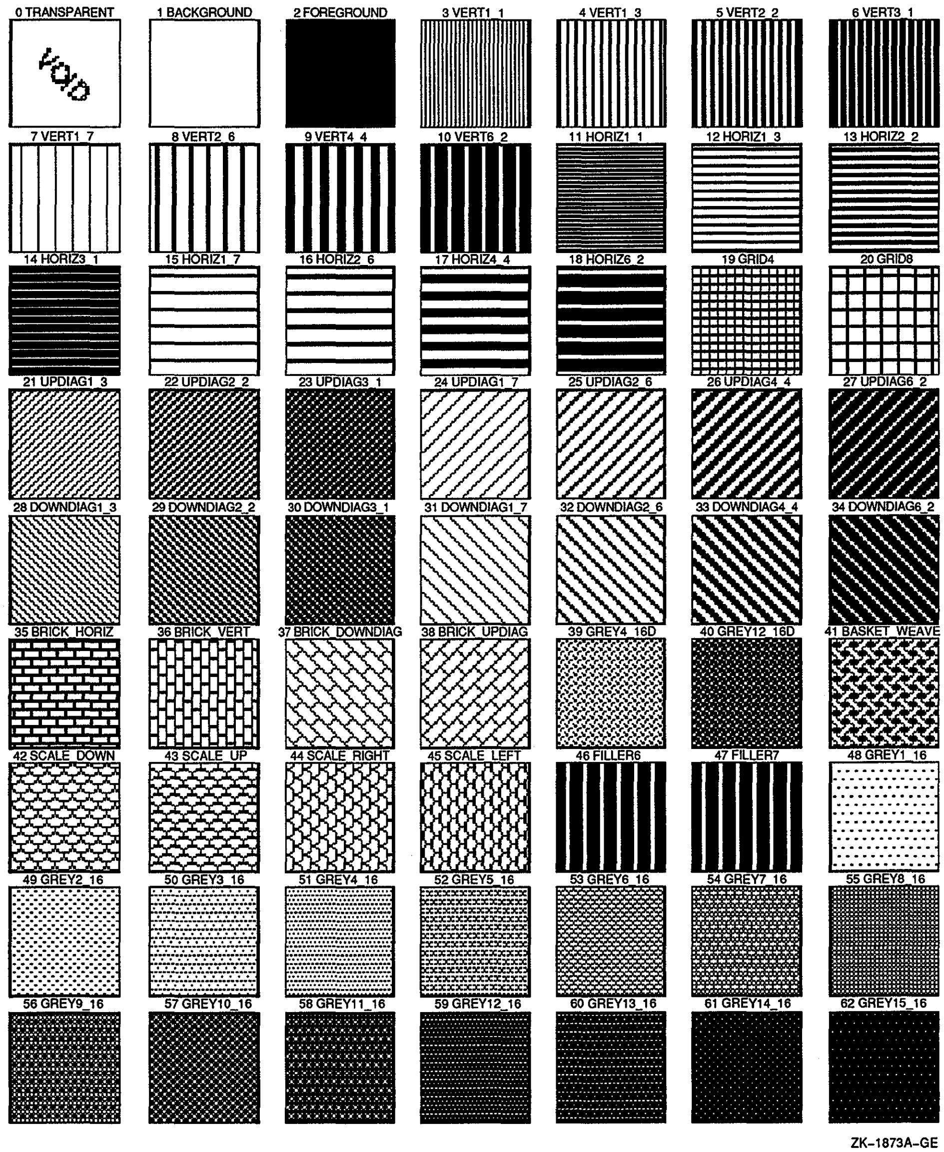
| Pattern Name | Number | Description |
|---|---|---|
| DDIF$K_PATT_DOWNDIAG6_2 | 34 | The thickness ratio of black to white downward diagonal lines (going down from left to right) in the pattern is 6 : 2. |
| DDIF$K_PATT_BRICK_HORIZ | 35 | The pattern is composed of bricks oriented in a horizontal direction. |
| 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). |
| 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 dots to the total number of dots in the pattern is 4 : 16. |
| DDIF$K_PATT_GREY12_16D | 40 | The ratio of black dots to the total number of dots in the pattern is 12 : 16. |
| DDIF$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$K_PATT_GREY1_16 | 48 | The ratio of black dots to the total number of dots in the pattern is 1 : 16. |
| DDIF$K_PATT_GREY2_16 | 49 | The ratio of black dots to the total number of dots in the pattern is 2 : 16. |
| DDIF$K_PATT_GREY3_16 | 50 | The ratio of black dots to the total number of dots in the pattern is 3 : 16. |
| DDIF$K_PATT_GREY4_16 | 51 | The ratio of black dots to the total number of dots in the pattern is 4 : 16. |

(continued on next page)

**Table A–1 (Cont.): DDIF Fill Patterns**

| Pattern Name | Number | Description |
|---|---|---|
| DDIF$K_PATT_GREY5_16 | 52 | The ratio of black dots to the total number of dots in the pattern is 5 : 16. |
| DDIF$K_PATT_GREY6_16 | 53 | The ratio of black dots to the total number of dots in the pattern is 6 : 16. |
| DDIF$K_PATT_GREY7_16 | 54 | The ratio of black dots to the total number of dots in the pattern is 7 : 16. |
| DDIF$K_PATT_GREY8_16 | 55 | The ratio of black dots to the total number of dots in the pattern is 8 : 16. |
| DDIF$K_PATT_GREY9_16 | 56 | The ratio of black dots to the total number of dots in the pattern is 9 : 16. |
| DDIF$K_PATT_GREY10_16 | 57 | The ratio of black dots to the total number of dots in the pattern is 10 : 16. |
| DDIF$K_PATT_GREY11_16 | 58 | The ratio of black dots to the total number of dots in the pattern is 11 : 16. |
| DDIF$K_PATT_GREY12_16 | 59 | The ratio of black dots to the total number of dots in the pattern is 12 : 16. |
| DDIF$K_PATT_GREY13_16 | 60 | The ratio of black dots to the total number of dots in the pattern is 13 : 16. |
| DDIF$K_PATT_GREY14_16 | 61 | The ratio of black dots to the total number of dots in the pattern is 14 : 16. |
| DDIF$K_PATT_GREY15_16 | 62 | The ratio of black dots to the total number of dots in the pattern is 15 : 16. |

## Figure A–1: CDA Fill Patterns



# Appendix B

# DDIF Syntax Diagrams

This appendix lists the syntax diagrams for each construct defined by DDIF (DIGITAL Document Interchange Format). The diagram for each construct is listed alphabetically under Syntax diagrams in the index. For example, Figure B–6 shows the syntax used to create a DDIF document construct and is listed as DDIFDocument under Syntax diagrams in the index.

The abstract syntax notation used to define these constructs at the lowest level is DDIS (DIGITAL Data Interchange Syntax). The elements of the DDIS abstract syntax notation that are used in this appendix are summarized in the following sections.

## B.1 DDIS Built-In Data Types

Table B–1 lists the built-in types that are primitive data types.

**Table B–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 B–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 DDIS. 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. Refer to the description of the corresponding DDIF$_EXT aggregate in Chapter 4. |

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 B–2 shows the DDIS constructors used in the DDIF syntax diagrams.

**Table B–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.

## B.2 Built-In Operators

Table B–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 B–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. |
| 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 represented by identifier (number). 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. |

## B.3 DDIS Defined Types

Table B–4 shows the types defined by DDIS.

**Table B–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. Refer to the description of the DDIF$_TXT aggregate in Chapter 4. |
| 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. |
| ObjectDescriptor | A type that models human-readable text that describes a data value. This text need not be unique within any context, as it is purely descriptive. |

Figure B–1 illustrates the syntax used to create an object descriptor construct.

**Figure B–1: Object Descriptor Syntax Diagram**

```
ObjectDescriptor ::= [UNIVERSAL 7] IMPLICIT OCTET STRING
```

Figure B–2 illustrates the syntax used to create a Latin1 construct.

**Figure B–2: Latin1 String Syntax Diagram**

```
Latin1-String ::= [PRIVATE 20] IMPLICIT OCTET STRING
```

Figure B–3 illustrates the syntax used to create a text string construct.

**Figure B–3: Text String Syntax Diagram**

```
Text-String ::=[ PRIVATE 10 ] IMPLICIT SEQUENCE OF Character-String
```

Figure B–4 illustrates the syntax used to create a character string construct.

**Figure B–4: Character String Syntax Diagram**

```
Character-String ::= [PRIVATE 9] IMPLICIT OCTET STRING
```

Figure B–5 illustrates the syntax used to create an application private data construct.

**Figure B–5: Application Private Data Syntax Diagram**

```
ApplPrivate ::= NamedValueList
```

# B.4 DDIF Syntax Diagrams

This section lists all the syntax diagrams that are used to describe DDIF constructs. Figure B–6 illustrates the syntax used to create a DDIF document construct.

Refer to the description of the corresponding DDIF$_DDF aggregate in Chapter 4.

**Figure B–6: 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 B–7 illustrates the syntax used to create a document descriptor construct.

Refer to the description of the corresponding DDIF$_DSC aggregate in Chapter 4.

**Figure B–7: 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 B–8 illustrates the syntax used to create a document header construct.

Refer to the description of the corresponding DDIF$_DHD aggregate in Chapter 4.

**Figure B–8: 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 B–9 illustrates the syntax used to create a document root segment construct.

**Figure B–9: 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 B–10 illustrates the syntax used to create a segment primitive construct.

**Figure B–10: Segment Primitive Syntax Diagram**

```
SegmentPrimitive                  ::= CHOICE {
     end-segment                       [APPLICATION 1] IMPLICIT NULL,
     begin-segment                     [APPLICATION 2] IMPLICIT BeginSegment
                                       }
```

Figure B–11 illustrates the syntax used to create a construct.

Refer to the description of the corresponding DDIF$_SEG aggregate in Chapter 4.

**Figure B–11: 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] ANY                        OPTIONAL,
     specific-layout              [5] ANY                        OPTIONAL
                                  }
```

Figure B–12 illustrates the syntax used to create a text primitive construct.

Refer to the description of the corresponding DDIF$_GTX aggregate in Chapter 4.

**Figure B–12: Text Primitive Syntax Diagram**

```
TextPrimitive                ::= CHOICE {
     latin1-content               [APPLICATION 3] IMPLICIT Latin1-String,
     general-text-content         [APPLICATION 4] IMPLICIT Character-String
                                  }
```

Figure B–13 illustrates the syntax used to create a text attributes construct.

Refer to the description of the corresponding Text Attributes, Text Font Attribute, Text Rendition Attribute, Text Size Attribute, the Text Direction Attribute, the Text Character Decimal Alignment Attribute, the Text Leader Attributes, and to the Text Kerning Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–13: 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 B–14 illustrates the syntax used to create a rendition code construct.

Refer to the description of the corresponding Text Rendition Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–14: 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),
     ideogram-overlined(62),
     ideogram-db-overlined(63),
     ideogram-stress-mark(64)            }
```

Figure B–15 illustrates the syntax used to create a leader style construct.

Refer to the description of the corresponding Text Leader Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–15: 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 B–16 illustrates the syntax used to create a text layout construct.

Refer to the description of the corresponding Layout Attribute, to the Galley-Based Layout Attribute, to the Path-Based Layout Attribute, to the Position-Relative Layout Attribute, and to the Text Position Attribute for the DDIF$_SGA aggregate in Chapter 4

**Figure B–16: Text Layout Syntax Diagram**

```
TextLayout            ::= CHOICE {
     galley-based-layout      [0] IMPLICIT SEQUENCE {
          wrap-attributes             [0] ANY OPTIONAL,
          galley-layout               [1] ANY OPTIONAL
                                      },
     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 B–17 illustrates the syntax used to create a text string layout construct.

Refer to the description of the corresponding Path-Based Layout Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–17: 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 B–18 illustrates the syntax used to create a formatting primitive construct.

Refer to the description of the corresponding DDIF$_HRD aggregate, DDIF$_HRV aggregate, DDIF$_SFT aggregate, and DDIF$_SFV aggregate in Chapter 4.

**Figure B–18: 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 B–19 illustrates the syntax used to create a value directive construct.

Refer to the description of the corresponding DDIF$_HRV aggregate in Chapter 4.

**Figure B–19: Value Directive Syntax Diagram**

```
ValueDirective              ::= CHOICE {
     escapement-directive       [0] IMPLICIT EscapementDirective,
     variable-reset             [1] IMPLICIT VariableReset
                                }
```

Figure B–20 illustrates the syntax used to create a directive construct.

Refer to the description of the corresponding DDIF$_HRD aggregate, the DDIF$_LL1 aggregate, and the DDIF$_SFT aggregate in Chapter 4.

**Figure B–20: 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 B–21 illustrates the syntax used to create an escapement directive construct.

Refer to the description of the corresponding DDIF$_HRV aggregate in Chapter 4.

**Figure B–21: Escapement Directive Syntax Diagram**

```
EscapementDirective          ::= Escapement
```

Figure B–22 illustrates the syntax used to create a variable reset construct.

Refer to the description of the corresponding DDIF$_HRV aggregate in Chapter 4.

**Figure B–22: Variable Reset Syntax Diagram**

```
VariableReset ::= SEQUENCE {
     reset-variable      [0] IMPLICIT VariableLabel,
     reset-value         [1] Expression
                         }
```

Figure B–23 illustrates the syntax used to create a graphics primitive construct.

**Figure B–23: 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 B–24 illustrates the syntax used to create a polyline construct.

Refer to the description of the corresponding DDIF$_LIN aggregate in Chapter 4.

**Figure B–24: 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
                                }
```

Figure B–25 illustrates the syntax used to create a cubic Bézier construct.

Refer to the description of the corresponding DDIF$_BEZ aggregate in Chapter 4.

**Figure B–25: 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 B–26 illustrates the syntax used to create an arc construct.

Refer to the description of the corresponding DDIF$_ARC aggregate in Chapter 4.

**Figure B–26: 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 B–27 illustrates the syntax used to create a fill area set construct.

Refer to the description of the corresponding DDIF$_FAS aggregate in Chapter 4.

Figure B–28 illustrates the syntax used to create a line attributes construct.

**Figure B–27: 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
                              }
```

Refer to the description of the corresponding Line Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–28: 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 B–29 illustrates the syntax used to create a line style number construct.

Refer to the description of the Line Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–29: Line Style Number Syntax Diagram**

```
LineStyleNumber             ::= INTEGER {
     solid(1),
     dash(2),
     dot(3),
     dash-dot(4)
                                        }
```

Figure B–30 illustrates the syntax used to create a line end number construct.

Refer to the description of the corresponding Line Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–30: Line End Number Syntax Diagram**

```
LineEndNumber               ::= INTEGER {
     butt-line-end(1),
     round-line-end(2),
     square-line-end(3),
     arrow(4)                           }
```

Figure B–31 illustrates the syntax used to create a line join construct.

Refer to the description of the corresponding Line Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–31: Line Join Syntax Diagram**

```
LineJoin                    ::= INTEGER {
     mitered-line-join(1),
     rounded-line-join(2),
     beveled-line-join(3)
                                        }
```

Figure B–32 illustrates the syntax used to create a marker attributes construct.

Refer to the description of the corresponding Marker Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–32: 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 B–33 illustrates the syntax used to create a marker number construct.

Refer to the description of the corresponding Marker Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–33: Marker Number Syntax Diagram**

```
MarkerNumber                ::= INTEGER {
     marker-dot(1),
     marker-plus-sign(2),
     marker-asterisk(3),
     marker-circle(4),
     marker-diagonal-cross(5)
     }
```

Figure B–34 illustrates the syntax used to create an image primitive construct.

Refer to the description of the corresponding DDIF$_IMG aggregate and to the DDIF$_IDU aggregate in Chapter 4.

**Figure B–34: 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 B–35 illustrates the syntax used to create an image coding attributes construct.

Refer to the description of the corresponding DDIF$_IDU aggregate and to the DDIF$_IMG aggregate in Chapter 4.

**Figure B–35: 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 B–36 illustrates the syntax used to create an image attributes construct.

Refer to the description of the corresponding Image Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–36: 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),    -- lookup table map
          rgb-mapped (5),    -- red-green-blue
          cmy-mapped (6),    -- cyan-magenta-yellow
          yuv-mapped (7),
          hsv-mapped (8),    -- hue saturation value
          hls-mapped (9),    -- hue lightness value
          yiq-mapped (10) }       OPTIONAL,
```

(continued on next page)

**Figure B–36 (Cont.): Image Attributes Syntax Diagram**

```
    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 B–37 illustrates the syntax used to create an image lookup table data construct.

Refer to the description of the corresponding Image Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–37: 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 B–38 illustrates the syntax used to create an image component space attributes construct.

Refer to the description of the corresponding Image Component Space Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–38: 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 B–39 illustrates the syntax used to create a restricted content construct.

**Figure B–39: Restricted Content Syntax Diagram**

```
RestrictedContent ::= CHOICE {
    pdl-content       [APPLICATION 18] IMPLICIT EXTERNAL,
    private-content   [APPLICATION 30] IMPLICIT NamedValue
                      }
```

Figure B–40 illustrates the syntax used to create a content reference primitive construct.

**Figure B–40: Content Reference Primitive Syntax Diagram**

```
ContentReferencePrimitive ::= CHOICE {
     content-ref                [APPLICATION 15] IMPLICIT ContentReference
                                }
```

Figure B–41 illustrates the syntax used to create a content reference construct.

Refer to the description of the corresponding DDIF$_CRF aggregate in Chapter 4.

**Figure B–41: Content Reference Syntax Diagram**

```
ContentReference          ::= SEQUENCE {
     content-transform          [0] IMPLICIT Transformation OPTIONAL,
     content-reference          [1] IMPLICIT ContentDefnLabel
                                }
```

Figure B–42 illustrates the syntax used to create a bounding box construct.

Refer to the description of the corresponding DDIF$_GLY aggregate and to the Frame Bounding Box Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–42: Bounding Box Syntax Diagram**

```
BoundingBox                ::= SEQUENCE {
     lower-left                  [0] IMPLICIT Position,
     upper-right                 [1] IMPLICIT Position
                                 }
```

Figure B–43 illustrates the syntax used to create a color construct.

Refer to the description of the corresponding DDIF$_PTD aggregate in Chapter 4.

**Figure B–43: Color Syntax Diagram**

```
Color                     ::= CHOICE {
    rgb-color                   [0] IMPLICIT RGB,
    transparency                [1] IMPLICIT NULL
                                }
```

Figure B–44 illustrates the syntax used to create a red/green/blue construct.

Refer to the description of the corresponding DDIF$_PTD aggregate and to the DDIF$_RGB aggregate in Chapter 4.

**Figure B–44: 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 B–45 illustrates the syntax used to create a compute definition construct.

Refer to the description of the Computed Content Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–45: 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 B–46 illustrates the syntax used to create a cross-reference construct.

Refer to the description of the corresponding Cross-Reference Computed Content Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–46: Cross-Reference Syntax Diagram**

```
CrossReference              ::= SEQUENCE {
     xref-seg-label              [0] IMPLICIT Reference,
     xref-var-label              [1] IMPLICIT VariableLabel
                                 }
```

Figure B–47 illustrates the syntax used to create an escapement construct.

Refer to the description of the DDIF$_HRV aggregate, the DDIF$_LL1 aggregate, and to the Position-Relative Layout Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–47: Escapement Syntax Diagram**

```
Escapement                  ::= SEQUENCE {
     escapement-ratio            [0] IMPLICIT Ratio OPTIONAL,
     escapement-constant         [1] Measurement    OPTIONAL
                                 }
```

Figure B–48 illustrates the syntax used to create an external reference construct.

Refer to the description of the corresponding DDIF$_ERF aggregate and the DDIF$_DHD aggregate in Chapter 4.

**Figure B–48: 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 B–49 illustrates the syntax used to create a font definition construct.

Refer to the description of the corresponding DDIF$_FTD aggregate and to the Font Definitions Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–49: Font Definition Syntax Diagram**

```
FontDefn                ::= SEQUENCE {
     font-number              [0] IMPLICIT FontNumber,
     font-identifier          [1] IMPLICIT Latin1-String,
     font-private             [2] IMPLICIT NamedValueList OPTIONAL
                              }
```

Figure B–50 illustrates the syntax used to create a format construct.

Refer to the description of the corresponding DDIF$_LW1 aggregate, the Path-Based Layout Attribute, the Frame Position Attribute, and to the Galley Frame Parameters for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–50: Format Syntax Diagram**

```
Format        ::= INTEGER { flush-path-begin(1), center-of-path(2),
                            flush-path-end(3), flush-path-both(4)
                            }
```

Figure B–51 illustrates the syntax used to create a frame parameters construct.

Refer to the description of the corresponding Frame Parameters Attributes, the Frame Flags Attribute, the Frame Bounding Box Attribute, the Frame Outline Attribute, the Frame Clipping Attribute, the Frame Position Attribute, the Fixed Frame Parameters, the Inline Frame Parameters, the Galley Frame Parameters, the Margin Frame Parameters, and to the Frame Content Transformation Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–51: 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 B–52 illustrates the syntax used to create an inline frame parameters construct.

**Figure B–52: Inline Frame Parameters Syntax Diagram**

```
InlineFrameParams          ::= SEQUENCE {
     ifp-base-offset            [0] Size DEFAULT { integer-constant 0 }
                                }
```

Refer to the description of the corresponding Frame Position Attribute and to the Inline Frame Parameters for the DDIF$_SGA aggregate in Chapter 4.

Figure B–53 illustrates the syntax used to create a galley frame parameters construct.

Refer to the description of the corresponding Frame Position Attribute and to the Galley Frame Parameters for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–53: 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 B–54 illustrates the syntax used to create a galley vertical position construct.

Refer to the description of the corresponding Frame Position Attribute and to the Galley Frame Parameters for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–54: Galley Vertical Position Syntax Diagram**

```
GalleyVerticalPosition ::= INTEGER {
     below-current-line(1),
     bottom-of-galley(2),
     top-of-galley(3)
     }
```

Figure B–55 illustrates the syntax used to create a margin frame parameters construct.

Refer to the description of the corresponding Frame Position Attribute and to the Margin Frame Parameters for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–55: 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 B–56 illustrates the syntax used to create a margin horizontal position construct.

Refer to the description of the corresponding Frame Position Attribute and to the Margin Frame Parameters for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–56: Margin Horizontal Position Syntax Diagram**

```
MarginHorizontalPosition ::= INTEGER {
     side-closest-edge(1),
     side-furthest-edge(2),
     left-of-galleys(3),
     right-of-galleys(4)
     }
```

Figure B–57 illustrates the syntax used to create a function link construct.

Refer to the description of the corresponding Function Computed Content Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–57: Function Link Syntax Diagram**

```
FunctionLink                  ::= SEQUENCE {
     function-name                 [0] IMPLICIT ASCIIString,
     function-parameters           [1] IMPLICIT NamedValueList
                                   }
```

Figure B–58 illustrates the syntax used to create an external reference index construct.

Refer to the description of the corresponding Copied and Remote Computed Content Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–58: External Reference Index Syntax Diagram**

```
ExternalRefIndex ::= INTEGER
```

Figure B–59 illustrates the syntax used to create a language index construct.

Refer to the description of the corresponding Language Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–59: Language Index Syntax Diagram**

```
LanguageIndex ::= INTEGER
```

Figure B–60 illustrates the syntax used to create a content definition construct.

Refer to the description of the corresponding DDIF$_CTD aggregate and to the Content Definitions Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–60: 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 B–61 illustrates the syntax used to create a label types construct.

Refer to the description of the corresponding DDIF$_CRF aggregate, the Variable Computed Content Attribute for the DDIF$_SGA aggregate, and to the DDIF$_TYD aggregate in Chapter 4.

**Figure B–61: Label Types Syntax Diagram**

```
VariableLabel              ::= Label

SegmentLabel               ::= Label

TypeDefnLabel              ::= Label

ContentDefnLabel           ::= Label

GalleyLabel                ::= Label

PageDescLabel              ::= Label

PageLayoutLabel            ::= Label
```

Figure B–62 illustrates the syntax used to create a label construct.

**Figure B–62: Label Syntax Diagram**

```
Label                    ::= ASCIIString
```

Figure B–63 illustrates the syntax used to create an ASCII string construct.

**Figure B–63: ASCII String Syntax Diagram**

```
ASCIIString ::= Latin1-String
```

Figure B–64 illustrates the syntax used to create a variable label construct.

**Figure B–64: Variable Label Syntax Diagram**

```
VariableLabel            ::= Label        -- used to refer to variable by name
```

Figure B–65 illustrates the syntax used to create a legend units construct.

Refer to the description of the corresponding Legend Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–65: Legend Units Syntax Diagram**

```
LegendUnits                  ::= SEQUENCE {
     legend-unit                   [0] IMPLICIT Ratio,
     legend-unit-name              [1] IMPLICIT Text-String
                                   }
```

Figure B–66 illustrates the syntax used to create an angle construct.

Refer to the description of the corresponding DDIF$_TRN aggregate, the Path-Based Layout Attribute, and to the Frame Content Transformation Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–66: Angle Syntax Diagram**

```
Angle                    ::= FLOATING-POINT
```

Figure B–67 illustrates the syntax used to create an AngleRef construct.

Refer to the description of the corresponding DDIF$_ARC aggregate and to the DDIF$_PTH aggregate in Chapter 4.

**Figure B–67: AngleRef Syntax Diagram**

```
AngleRef                 ::= CHOICE {
     angle-constant           [0] IMPLICIT Angle,
     angle-variable           [1] IMPLICIT VariableLabel
                              }
```

Figure B–68 illustrates the syntax used to create a measurement construct.

Refer to the description of the corresponding DDIF$_PTH aggregate and to the Frame Position Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–68: Measurement Syntax Diagram**

```
Measurement                ::= CHOICE {
     integer-constant           [0] IMPLICIT INTEGER,
     variable-measure           [1] IMPLICIT VariableLabel
                                }
```

Figure B–69 illustrates the syntax used to create a position construct.

Refer to the description of the corresponding Frame Bounding Box Attributes, the Frame Position Attribute, and to the Fixed Frame parameters for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–69: Position Syntax Diagram**

```
Position                 ::= SEQUENCE {
     x-coordinate             [0] XCoordinate,
     y-coordinate             [1] YCoordinate
                              }
```

Figure B–70 illustrates the syntax used to create a ratio construct.

Refer to the description of the corresponding DDIF$_HRV aggregate and to the Line Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–70: Ratio Syntax Diagram**

```
Ratio                        ::= SEQUENCE {
  numerator                        [0] IMPLICIT INTEGER DEFAULT 1,
  denominator                      [1] IMPLICIT INTEGER DEFAULT 100
                                   }
```

Figure B–71 illustrates the syntax used to create a right angle construct.

Refer to the description of the corresponding Path-Based Layout Attribute for the DDIF$_SGA aggregate in Chapter 4.

Figure B–72 illustrates the syntax used to create a size construct.

Refer to the description of the corresponding DDIF$_ARC aggregate and to the Text Size Attribute, the Frame Position Attribute, and the Inline Frame Parameters for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–71: Right Angle Syntax Diagram**

```
RightAngle  ::= INTEGER { right(1),
                          left(2),
                          up(3),
                          down(4)
                        }
```

**Figure B–72: Size Syntax Diagram**

```
Size                      ::= Measurement
```

Figure B–73 illustrates the syntax used to create an *x*-coordinate construct.

Refer to the description of the corresponding DDIF$_ARC aggregate, the DDIF$_PTH aggregate, and to the Frame Position Attribute and the Fixed Frame Parameters for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–73: X-Coordinate Syntax Diagram**

```
XCoordinate                 ::= Measurement
```

Figure B–74 illustrates the syntax used to create a *y*-coordinate construct.

Refer to the description of the corresponding DDIF$_ARC aggregate, the DDIF$_PTH aggregate, and to the Frame Position Attribute and the Fixed Frame Parameters for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–74: Y-Coordinate Syntax Diagram**

```
YCoordinate                 ::= Measurement
```

Figure B–75 illustrates the syntax used to create a measurement units construct.

Refer to the description of the corresponding Measurement Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–75: Measurement Units Syntax Diagram**

```
MeasurementUnits            ::= SEQUENCE {
    units-per-measurement         [0] IMPLICIT INTEGER,
    unit-name                     [1] IMPLICIT Text-String
                                  }
```

Figure B–76 illustrates the syntax used to create a named value construct.

**Figure B–76: Named Value Syntax Diagram**

```
NamedValue              ::= SEQUENCE {
   value-name                 NamedValueTag,
   value-data                 ValueData
                              }
```

Figure B–77 illustrates the syntax used to create a value data construct.

Refer to the description of the DDIF$_PVT aggregate in Chapter 4.

**Figure B–77: 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 B–78 illustrates the syntax used to create a named value list construct.

Refer to the description of the corresponding DDIF$_CTD aggregate, the DDIF$_FTD aggregate, the DDIF$_DHD aggregate, the DDIF$_IDU aggregate, the DDIF$_LG1 aggregate, the DDIF$_PGD aggregate, the DDIF$_PHD aggregate, the DDIF$_LSD aggregate, and to the DDIF$_TYD aggregate in Chapter 4.

**Figure B–78: Named Value List Syntax Diagram**

```
NamedValueList ::= SEQUENCE OF NamedValue
```

Figure B–79 illustrates the syntax used to create a font number construct.

Refer to the description of the corresponding Font Definitions Attribute and to the Text Font Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–79: Font Number Syntax Diagram**

```
FontNumber                 ::= INTEGER
```

Figure B–80 illustrates the syntax used to create a marker number construct.

**Figure B–80: Marker Number Syntax Diagram**

```
MarkerNumber                 ::= INTEGER
```

Figure B–81 illustrates the syntax used to create a path number construct.

**Figure B–81: Path Number Syntax Diagram**

```
PathNumber                ::= INTEGER
```

Refer to the description of the corresponding DDIF$_PTH aggregate in Chapter 4.

Figure B–82 illustrates the syntax used to create a pattern number construct.

Refer to the description of the corresponding Text Mask Pattern Attribute, the Line Attributes, and to the Marker Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–82: Pattern Number Syntax Diagram**

```
PatternNumber             ::= INTEGER
```

Figure B–83 illustrates the syntax used to create a path definition construct.

Refer to the description of the corresponding DDIF$_PHD aggregate and to the Path Definitions Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–83: Path Definition Syntax Diagram**

```
PathDefn                  ::= SEQUENCE  {
     path-number                [0] IMPLICIT PathNumber,
     path-description           [1] IMPLICIT CompositePath,
     path-private               [2] IMPLICIT NamedValueList          OPTIONAL
                                }
```

Figure B–84 illustrates the syntax used to create a composite path construct.

Refer to the description of the corresponding DDIF$_FAS aggregate, the DDIF$_GLY aggregate, the DDIF$_PHD aggregate, the DDIF$_PTH aggregate, and to the Frame Outline Attribute and Frame Clipping Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–84: 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 B–85 illustrates the syntax used to create an arc path construct.

Refer to the description of the corresponding DDIF$_ARC aggregate and to the DDIF$_PTH aggregate in Chapter 4.

**Figure B–85: 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 B–86 illustrates the syntax used to create a cubic Bézier path construct.

Refer to the description of the corresponding DDIF$_BEZ aggregate and to the DDIF$_PTH aggregate in Chapter 4.

**Figure B–86: Cubic Bézier Path Syntax Diagram**

```
CubicBezierPath            ::= SEQUENCE OF Measurement
```

Figure B–87 illustrates the syntax used to create a line definition construct.

Refer to the description of the corresponding DDIF$_LSD aggregate and to the Line-Style Definitions Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–87: 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 B–88 illustrates the syntax used to create a polyline path construct.

Refer to the description of the corresponding DDIF$_LIN aggregate and to the DDIF$_PTH aggregate in Chapter 4.

**Figure B–88: Polyline Path Syntax Diagram**

```
PolyLinePath              ::= SEQUENCE OF Measurement
```

Figure B–89 illustrates the syntax used to create a pattern definition construct.

Refer to the description of the corresponding DDIF$_PTD aggregate and to the Pattern Definitions Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–89: 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 B–90 illustrates the syntax used to create a standard pattern construct.

Refer to the description of the corresponding DDIF$_PTD aggregate in Chapter 4.

**Figure B–90: 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 B–91 illustrates the syntax used to create a reference construct.

Refer to the description of the corresponding DDIF$_PVT aggregate and to the Copied and Remote Computed Content Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–91: Reference Syntax Diagram**

```
Reference                   ::= SEQUENCE {
    ref-target                    [0] IMPLICIT SegmentLabel OPTIONAL,
    ref-x-index                   [1] IMPLICIT ExternalRefIndex OPTIONAL
                                  }
```

Figure B–92 illustrates the syntax used to create a segment attributes construct.

Refer to the description of the corresponding DDIF$_SEG aggregate, the DDIF$_TYD aggregate, the DDIF$_SGA aggregate, and General Segment Attributes and Alternate Presentation Attribute for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–92: 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] ANY                                OPTIONAL,

     image-attributes           [22] IMPLICIT ImageAttributes           OPTIONAL,
     frame-parameters           [23] IMPLICIT FrameParameters           OPTIONAL
                                }
```

Figure B–93 illustrates the syntax used to create a segment type definition construct.

Refer to the description of the corresponding Type Definitions Attribute for the DDIF$_SGA aggregate and to the DDIF$_TYD aggregate in Chapter 4.

**Figure B–93: 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 B–94 illustrates the syntax used to create a structure definition construct.

Refer to the description of the corresponding DDIF$_OCC aggregate and to the Structure Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–94: 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 B–95 illustrates the syntax used to create an occurrence definition construct.

Refer to the description of the corresponding DDIF$_OCC aggregate in Chapter 4.

**Figure B–95: Occurrence Definition Syntax Diagram**

```
OccurrenceDefn          ::= CHOICE {
    required-element            [0] StructureElement,
    optional-element            [1] StructureElement,
    repeat-element              [2] StructureElement,
    opt-repeat-element          [3] StructureElement
                                }
```

Figure B–96 illustrates the syntax used to create a structure element construct.

Refer to the description of the corresponding DDIF$_OCC aggregate in Chapter 4.

### Figure B–96: Structure Element Syntax Diagram

```
StructureElement            ::= CHOICE {
    expression-element            StructureDefn,
    referenced-type               TypeDefnLabel
                                  }
```

Figure B–97 illustrates the syntax used to create a tag construct.

### Figure B–97: Tag Syntax Diagram

```
Tag                      ::= ASCIIString
```

Figure B–98 illustrates the syntax used to create a category tag construct.

### Figure B–98: Category Tag Syntax Diagram

```
CategoryTag                 ::= Tag
```

Figure B–99 illustrates the syntax used to create a conformance tag construct.

### Figure B–99: Conformance Tag Syntax Diagram

```
ConformanceTag           ::= Tag
```

Figure B–100 illustrates the syntax used to create a named value tag construct.

### Figure B–100: Named Value Tag Syntax Diagram

```
NamedValueTag            ::= Tag
```

Figure B–101 illustrates the syntax used to create a segment tag construct.

Refer to the description of the corresponding Counter Variable Values for the DDIF$_SGB aggregate in Chapter 4.

**Figure B–101: Segment Tag Syntax Diagram**

```
SegmentTag              ::= Tag
```

Figure B–102 illustrates the syntax used to create a storage system tag construct.

**Figure B–102: Storage System Tag Syntax Diagram**

```
StorageSystemTag        ::= Tag
```

Figure B–103 illustrates the syntax used to create a stream tag construct.

**Figure B–103: Stream Tag Syntax Diagram**

```
StreamTag               ::= Tag
```

Figure B–104 illustrates the syntax used to create a transformation construct.

Refer to the description of the corresponding DDIF$_CRF aggregate, the Frame Content Transformation Attribute for the DDIF$_SGA aggregate, and to the DDIF$_TRN aggregate in Chapter 4.

**Figure B–104: 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 B–105 illustrates the syntax used to create a variable binding construct.

Refer to the description of the corresponding DDIF$_SGB aggregate and to the Counter Variable Values, Computed Variable Values, and to the List Variable Values for the DDIF$_SGB aggregate in Chapter 4.

**Figure B–105: 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 B–106 illustrates the syntax used to create a counter definition construct.

Refer to the description of the corresponding DDIF$_SGB aggregate and to the Counter Variable Values for the DDIF$_SGB aggregate in Chapter 4.

**Figure B–106: 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 B–107 illustrates the syntax used to create a layout object type construct.

Refer to the description of the corresponding Counter Variable Values for the DDIF$_SGB aggregate in Chapter 4.

**Figure B–107: 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 B–108 illustrates the syntax used to create an expression construct.

Refer to the description of the corresponding Counter Variable Values for the DDIF$_SGB aggregate in Chapter 4.

**Figure B–108: Expression Syntax Diagram**

```
Expression                ::= CHOICE {
     exp-integer               [0] IMPLICIT INTEGER,
     exp-variable              [1] IMPLICIT VariableLabel
                               }
```

Figure B–109 illustrates the syntax used to create a counter style construct.

Refer to the description of the corresponding DDIF$_CTS aggregate and to the Counter Variable Values for the DDIF$_SGB aggregate in Chapter 4.

**Figure B–109: 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 B–110 illustrates the syntax used to create a string expression construct.

Refer to the description of the corresponding DDIF$_SGB aggregate and to the Computed Variable Values for the DDIF$_SGB aggregate in Chapter 4.

**Figure B–110: String Expression Syntax Diagram**

```
StringExpression          ::= SEQUENCE OF CHOICE {
     text-element              [0] IMPLICIT Character-String,
     variable-ref-element      [1] IMPLICIT VariableLabel
                               }
```

Figure B–111 illustrates the syntax used to create a record list construct.

Refer to the description of the corresponding DDIF$_SGB aggregate and to the List Variable Values for the DDIF$_SGB aggregate in Chapter 4.

**Figure B–111: Record List Syntax Diagram**

```
RecordList ::= SEQUENCE OF RecordDefn
```

Figure B–112 illustrates the syntax used to create a record definition construct.

Refer to the description of the corresponding DDIF$_RCD aggregate and to the List Variable Values for the DDIF$_SGB aggregate in Chapter 4.

**Figure B–112: Record Definition Syntax Diagram**

```
RecordDefn              ::= SEQUENCE {
     record-type             [0] IMPLICIT TypeDefnLabel,
     record-tag              [1] IMPLICIT SegmentTag,
     record-contents         [2] IMPLICIT SEQUENCE OF VariableLabel
                             }
```

Figure B–113 illustrates the syntax used to create a generic layout construct.

Refer to the description of the corresponding DDIF$_LG1 aggregate in Chapter 4.

**Figure B–113: 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 B–114 illustrates the syntax used to create a page description construct.

Refer to the description of the corresponding DDIF$_LG1 aggregate, the DDIF$_LS1 aggregate, and to the DDIF$_PGD aggregate in Chapter 4.

**Figure B–114: 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 B–115 illustrates the syntax used to create a page set construct.

Refer to the description of the corresponding DDIF$_PGD aggregate and to the DDIF$_PGS aggregate in Chapter 4.

## Figure B–115: 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 B–116 illustrates the syntax used to create a page layout construct.

Refer to the description of the corresponding DDIF$_PGD aggregate, the DDIF$_PGL aggregate, and to the DDIF$_PGS aggregate in Chapter 4.

## Figure B–116: 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 B–117 illustrates the syntax used to create a layout primitive construct.

## Figure B–117: Layout Primitive Syntax Diagram

```
LayoutPrimitive          ::= [APPLICATION 35] ANY
```

Figure B–118 illustrates the syntax used to create a layout galley construct.

Refer to the description of the corresponding DDIF$_GLY aggregate in Chapter 4.

**Figure B–118: 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 B–119 illustrates the syntax used to create a galley attributes construct.

Refer to the description of the corresponding DDIF$_GLA aggregate and to the Galley Attributes for the DDIF$_SGA aggregate in Chapter 4.

**Figure B–119: 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 B–120 illustrates the syntax used to create a specific layout construct.

Refer to the description of the corresponding DDIF$_LS1 aggregate in Chapter 4.

**Figure B–120: Specific Layout Syntax Diagram**

```
SpecificLayout         ::= [APPLICATION 32] IMPLICIT SEQUENCE OF CHOICE {
     specific-page          [0] IMPLICIT PageDescription,
     referenced-page        [1] IMPLICIT PageDescLabel
                            }
```

Figure B–121 illustrates the syntax used to create a wrap attributes construct.

Refer to the description of the corresponding DDIF$_LW1 aggregate in Chapter 4.

**Figure B–121: 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 B–122 illustrates the syntax used to create a layout attributes construct.

Refer to the description of the corresponding DDIF$_LL1 aggregate in Chapter 4.

**Figure B–122: 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 B–123 illustrates the syntax used to create a break criteria construct.

Refer to the description of the corresponding DDIF$_LL1 aggregate in Chapter 4.

**Figure B–123: Break Criteria Syntax Diagram**

```
BreakCriteria               ::= INTEGER {
     break-always(1),
     break-never(2),
     break-if-needed(3)                  }
```

Figure B–124 illustrates the syntax used to create a general measure construct.

Refer to the description of the corresponding DDIF$_PGL aggregate in Chapter 4.

**Figure B–124: 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 B–125 illustrates the syntax used to create a general size construct.

Refer to the description of the corresponding DDIF$_PGL aggregate in Chapter 4.

**Figure B–125: General Size Syntax Diagram**

```
GenSize                     ::= SEQUENCE {
     x-size                      [0] IMPLICIT GenMeasure,
     y-size                      [1] IMPLICIT GenMeasure
                                 }
```

Figure B–126 illustrates the syntax used to create a tab stop list construct.

Refer to the description of the corresponding DDIF$_LL1 aggregate in Chapter 4.

**Figure B–126: Tab Stop List Syntax Diagram**

```
TabStopList                 ::= SEQUENCE OF TabStop
```

Figure B–127 illustrates the syntax used to create a tab stop construct.

Refer to the description of the corresponding DDIF$_TBS aggregate in Chapter 4.

**Figure B–127: 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
                                 }
```

Figure B–128 illustrates the syntax used to create a generalized time construct.

Refer to the description of the corresponding DDIF$_DHD aggregate in Chapter 4.

**Figure B–128: Generalized Time Diagram**

```
GeneralizedTime          ::= [UNIVERSAL 24] IMPLICIT OCTET STRING
```

# Appendix C

# DTIF Syntax Diagrams

This appendix lists the syntax diagrams for each construct defined by DTIF (DIGITAL Table Interchange Format). The diagram for each construct is listed alphabetically under Syntax diagrams in the index. For example, Figure C–1 shows the syntax used to create a DTIF document construct and is listed as DTIFDocument under Syntax diagrams in the index. For a description of the DDIS types referred to in the syntax diagrams, see Appendix B.

Figure C–1 illustrates the syntax used to create a DTIF document construct.

Refer to the description of the corresponding DTIF$_DTF aggregate in Chapter 5.

**Figure C–1: DTIF Document Syntax Diagram**

```
DTIFDocument ::= [PRIVATE 16382] IMPLICIT SEQUENCE {
    document-descriptor   [0] IMPLICIT DocumentDescriptor,
    document-header       [1] IMPLICIT DocumentHeader,
    document-tables       [3] IMPLICIT SEQUENCE OF TableDefn
                          }
```

Figure C–2 illustrates the syntax used to create a document descriptor construct.

Refer to the description of the corresponding DTIF$_DSC aggregate in Chapter 5.

**Figure C–2: Document Descriptor Syntax Diagram**

```
DocumentDescriptor ::= SEQUENCE {
    major-version         [0] IMPLICIT INTEGER,     -- product version
    minor-version         [1] IMPLICIT INTEGER,     -- product version
    product-identifier    [2] IMPLICIT ASCIIString,
    product-name          [3] IMPLICIT Text-String,
    encode-major-version  [4] IMPLICIT INTEGER,      -- DTIF encoding version
    encode-minor-version  [5] IMPLICIT INTEGER       -- DTIF encoding version
                          }
```

Figure C–3 illustrates the syntax used to create a document header construct.

Refer to the description of the corresponding DTIF$_HDR aggregate in Chapter 5.

**Figure C–3: Document Header Syntax Diagram**

```
DocumentHeader ::= SEQUENCE {
    private-header-data   [0] IMPLICIT NamedValueList OPTIONAL,
    title                 [1] IMPLICIT Text-String OPTIONAL,
    date                  [4] IMPLICIT DateTime    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,
    language-pref-tables  [9] IMPLICIT SEQUENCE OF LangPrefTable OPTIONAL,
    generic-columns      [10] IMPLICIT ColAttrList OPTIONAL
                          }
```

Figure C–4 illustrates the syntax used to create an external reference construct.

Refer to the description of the corresponding DTIF$_ERF aggregate in Chapter 5.

**Figure C–4: 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 C–5 illustrates the syntax used to create a storage system tag construct.

Refer to the description of the corresponding DTI$_ERF aggregate in Chapter 5.

**Figure C–5: Storage System Tag Syntax Diagram**

```
StorageSystemTag ::= ASCIIString
```

Figure C–6 illustrates the syntax used to create an external references index construct.

Refer to the description of the corresponding DTIF$_ERF aggregate in Chapter 5.

**Figure C–6: External References Index Syntax Diagram**

```
ExternalRefIndex  ::= INTEGER -- index into ExternalReferences
```

Figure C–7 illustrates the syntax used to create a language preference table construct.

Refer to the description of the corresponding DTIF$_LPT aggregate in Chapter 5.

**Figure C–7: Language Preference Table Syntax Diagram**

```
LangPrefTable  ::= SEQUENCE {
    pref-language-index  [0] IMPLICIT INTEGER OPTIONAL,
    pref-appl-priv       [1] IMPLICIT ApplPrivate     OPTIONAL,
    pref-items           [2] IMPLICIT NamedValueList  OPTIONAL,
    pref-editstrs        [3] IMPLICIT SEQUENCE OF NamedEditString OPTIONAL,
    pref-collate-seq     [4] IMPLICIT Latin1-String   OPTIONAL,
    pref-collate-table   [5] IMPLICIT OCTET STRING    OPTIONAL
                           }
```

Figure C–8 illustrates the syntax used to create a named edit string construct.

Refer to the description of the corresponding DTIF$_NES aggregate in Chapter 5.

**Figure C–8: Named Edit String Syntax Diagram**

```
NamedEditString ::= SEQUENCE {
    editstring-name [0] IMPLICIT ASCIIString  OPTIONAL,
    editstring-defn [1] EditString
                   }
```

Figure C–9 illustrates the syntax used to create a table definition construct.

Refer to the description of the corresponding DTIF$_TBL aggregate in Chapter 5.

**Figure C–9: Table Definition Syntax Diagram**

```
TableDefn          ::= SEQUENCE {
    table-max-cols      [0] IMPLICIT INTEGER OPTIONAL,
    table-max-rows      [1] IMPLICIT INTEGER OPTIONAL,
    table-appl-private  [2] IMPLICIT ApplPrivate OPTIONAL,
    table-metadata      [3] IMPLICIT TableMD OPTIONAL,
    table-windows       [4] IMPLICIT SEQUENCE OF WindowDefn OPTIONAL,
    table-rows          [5] IMPLICIT SEQUENCE OF RowDefn OPTIONAL
                        }
```

Figure C–10 illustrates the syntax used to create a table metadata construct.

Refer the description of the corresponding DTIF$_TMD aggregate in Chapter 5.

**Figure C–10: Table Metadata Syntax Diagram**

```
TableMD             ::= SEQUENCE {
    tmd-name            [0] IMPLICIT Text-String OPTIONAL,
    tmd-id              [1] IMPLICIT INTEGER OPTIONAL,
    tmd-appl-priv       [2] IMPLICIT ApplPrivate OPTIONAL,
    tmd-description     [3] IMPLICIT Text-String OPTIONAL,
    tmd-flags           [4] IMPLICIT BIT STRING {
                                     tmd-autorecalc  (0),
                                     tmd-autoresort  (1),
                                     tmd-calcbycol   (2),
                                     tmd-calcbyrow   (3),
                                     tmd-calcnatural (4),
                                     tmd-fmtbycol    (5),
                                     tmd-fmtbyrow    (6)
                                     } DEFAULT {tmd-fmtbycol},
    tmd-default-fmts    [5] IMPLICIT FormatInfoList OPTIONAL,
    tmd-columns         [6] IMPLICIT ColAttrList OPTIONAL,
    tmd-ranges          [7] IMPLICIT SEQUENCE OF RangeDefn OPTIONAL,
    tmd-symbols         [8] IMPLICIT NamedValueList OPTIONAL
                        }
```

Figure C–11 illustrates the syntax used to create a table window construct.

Refer to the description of the corresponding DTIF$_WND aggregate in Chapter 5.

**Figure C–11: Table Window Syntax Diagram**

```
WindowDefn          ::= SEQUENCE {
    wnd-name            [0] IMPLICIT Text-String OPTIONAL,
    wnd-id              [1] IMPLICIT INTEGER OPTIONAL,
    wnd-appl-priv       [2] IMPLICIT ApplPrivate OPTIONAL,
    wnd-cardinal-num    [3] IMPLICIT INTEGER OPTIONAL,
    wnd-description     [4] IMPLICIT Text-String OPTIONAL,
    wnd-flags           [5] IMPLICIT BIT STRING {
                                     window-active         (0),
                                     window-hidden         (1),
                                     window-formula-hidden (2),
                                     window-value-hidden   (3),
                                     window-colhdr-hidden  (4),
                                     window-rowhdr-hidden  (5),
                                     window-lines-hidden   (6)
                                     } DEFAULT {window-formula-hidden},
    wnd-formats         [6] IMPLICIT FormatInfoList OPTIONAL,
    wnd-ranges          [7] IMPLICIT SEQUENCE OF RangeDefn OPTIONAL,
    wnd-active-loc      [8] IMPLICIT CellCoord OPTIONAL

                        }
```

Figure C–12 illustrates the syntax used to create a table rows construct.

Refer to the description of the corresponding DTIF$_ROW aggregate in Chapter 5.

**Figure C–12: Table Rows Syntax Diagram**

```
RowDefn             ::= SEQUENCE {
    row-num             [0] IMPLICIT RowNum OPTIONAL,
    row-appl-priv       [1] IMPLICIT ApplPrivate  OPTIONAL,
    row-formats         [2] IMPLICIT FormatInfoList OPTIONAL,
    row-flags           [3] IMPLICIT BIT STRING {
                                  row-annotation (0)
                                  } OPTIONAL,
    row-cells           [4] IMPLICIT SEQUENCE OF CellData OPTIONAL
                        }
```

Figure C–13 illustrates the syntax used to create a cell data construct.

Refer to the description of the corresponding DTIF$_CLD aggregate in Chapter 5.

**Figure C–13: Cell Data Syntax Diagram**

```
CellData ::= SEQUENCE {
    cell-col-num        [0] IMPLICIT ColNum OPTIONAL,
    cell-state          [1] IMPLICIT INTEGER {
                                -- Basic cell states:
                                cs-isvalue   (0),
                                cs-isnull    (1),
                                cs-iserror   (2),
                                cs-isnovalue (3),

                                -- Additional Error States.
                                cs-isunderflow  (10),
                                cs-isoverflow   (11),
                                cs-isundefref   (12),
                                cs-isdivzero    (13),
                                cs-isrecursive  (14)
                                } DEFAULT {cs-isvalue},
    cell-description    [2] IMPLICIT Text-String OPTIONAL,
    cell-appl-priv      [3] IMPLICIT ApplPrivate OPTIONAL
    cell-formats        [4] IMPLICIT FormatInfoList OPTIONAL,
    cell-value          [5] CellValue  OPTIONAL,
    cell-formula-cfe    [6] Expression OPTIONAL
                        }
```

Figure C–14 illustrates the syntax used to create a cell value construct.

Refer to the description of the corresponding DTIF$_CLD aggregate in Chapter 5.

**Figure C–14: Cell Value Syntax Diagram**

```
CellValue         ::= CHOICE {
    cv-integer          [0] IMPLICIT INTEGER,
    cv-latin1-text      [1] IMPLICIT Latin1-String,
    cv-simple-text      [2] IMPLICIT Character-String,
    cv-date             [3] IMPLICIT DateTime,
    cv-scaled-integer   [5] IMPLICIT SCALED-INTEGER,
    cv-vtext            [6] IMPLICIT VaryingText,
    cv-array            [7] IMPLICIT ArrayDefn,
    cv-complex          [8] IMPLICIT ComplexFloat,
    cv-float            [9] IMPLICIT FLOATING-POINT,
    cv-boolean         [10] IMPLICIT BOOLEAN
                        }
```

Figure C–15 illustrates the syntax used to create a varying text construct.

Refer to the description of the corresponding DTIF$_VTX aggregate in Chapter 5.

**Figure C–15: Varying Text Syntax Diagram**

```
VaryingText ::= SEQUENCE {
    vtext-len    [0] IMPLICIT INTEGER,
    vtext-str    [1] IMPLICIT Character-String
                         }
```

Figure C–16 illustrates the syntax used to create an array definition construct.

Refer to the description of the corresponding DTIF$_ARD aggregate in Chapter 5.

**Figure C–16: Array Definition Syntax Diagram**

```
ArrayDefn        ::= SEQUENCE {
    array-description    [0] IMPLICIT Latin1-String OPTIONAL,
    array-elem-type-size [1] CHOICE {
               std-type [0] IMPLICIT INTEGER {
                                elem-word (0),
                                elem-long (1),
                                elem-ffloat (2),
                                elem-dfloat (3),
                                elem-gfloat (4),
                                elem-hfloat (5)
                                     },
               var-type [1] IMPLICIT INTEGER
                              },
     array-x-dimension    [2] IMPLICIT INTEGER,
     array-y-dimension    [3] IMPLICIT INTEGER OPTIONAL,
     array-z-dimension    [4] IMPLICIT INTEGER OPTIONAL,
     array-values         [5] IMPLICIT OCTET STRING
                          }
```

Figure C–17 illustrates the syntax used to create a complex floating-point construct.

Refer to the description of the corresponding DTIF$_CFT aggregate in Chapter 5, and to the CFE$_CFT aggregate and CFE$_EXL aggregate in Chapter 6 float construct.

**Figure C–17: Complex Float Syntax Diagram**

```
ComplexFloat  ::= SEQUENCE
    {
    real-part      [0] IMPLICIT FLOATING-POINT,
    imaginary-part [1] IMPLICIT FLOATING-POINT
    }
```

Figure C–18 illustrates the syntax used to create a column attributes construct.

Refer to the description of the corresponding DTIF$_CAT aggregate in Chapter 5.

**Figure C–18: Column Attributes Syntax Diagram**

```
ColAttrList ::= SEQUENCE OF ColAttributes

ColAttributes ::= SEQUENCE {
    col-name             [0] IMPLICIT ASCIIString OPTIONAL,
    col-id               [1] IMPLICIT INTEGER OPTIONAL,
    col-appl-priv        [2] IMPLICIT ApplPrivate OPTIONAL
    col-generic-ref      [3] IMPLICIT INTEGER OPTIONAL,
           -- not used in generic-columns;
           -- used by table-col to reference generic col-id
    col-description      [4] IMPLICIT Text-String OPTIONAL,
    col-formats          [5] IMPLICIT FormatInfoList OPTIONAL,
    col-computed-by      [6] Expression OPTIONAL,
    col-default-value    [7] CellValue  OPTIONAL,
    col-missing-value    [8] CellValue  OPTIONAL,
    col-query-name       [9] IMPLICIT Character-String   OPTIONAL,
    col-column-hdr      [10] IMPLICIT Character-String   OPTIONAL,
    col-data-type       [12] IMPLICIT Datatype OPTIONAL,
    col-data-length     [13] IMPLICIT INTEGER OPTIONAL,
    col-scale-factor    [14] IMPLICIT INTEGER OPTIONAL,
    col-flags           [15] IMPLICIT BIT STRING {
                                  col-autorecalc (0),
                                  col-readonly   (1),
                                  col-annotation (2)
                                  } OPTIONAL

                        }
```

Figure C–19 illustrates the syntax used to create a data type construct.

Refer to the description of the corresponding DTIF$_CAT aggregate in Chapter 5.

### Figure C–19: Data Type Syntax Diagram

```
Datatype    ::= INTEGER {
                   dt-unknown (0),
                   dt-word    (1),   --- signed word integer (16 bits)
                   dt-long    (2),   --- signed longword integer (32 bits)
                   dt-quad    (3),   --- signed quadword integer (64 bits)
                   dt-ffloat  (4),
                   dt-dfloat  (5),
                   dt-gfloat  (6),
                   dt-hfloat  (7),
                   dt-absdate (8),   --- absolute date/time
                   dt-text    (9),   --- text string
                   dt-vtext  (10),   --- varying text string
                   dt-segstr (11)    --- segmented string
                   }
```

Figure C–20 illustrates the syntax used to create a format information list construct.

Refer to the description of the corresponding DTIF$_FMI aggregate in Chapter 5.

### Figure C–20: Format Info List Syntax Diagram

```
FormatInfoList ::= SEQUENCE OF FormatInfo

FormatInfo ::= SEQUENCE {
    format-window-id    [0] IMPLICIT INTEGER OPTIONAL,
    format-type         [1] FormatType OPTIONAL,
    format-flags        [2] IMPLICIT FmtFlags OPTIONAL,
    format-width        [3] IMPLICIT INTEGER  OPTIONAL,
    format-lang-id      [4] IMPLICIT LangPrefIndex OPTIONAL,
    format-direction    [5] IMPLICIT INTEGER {
                                  dir-opposite (0)
                                } OPTIONAL,
    format-unit-desc    [6] IMPLICIT Text-String OPTIONAL,
    format-alignment    [7] IMPLICIT INTEGER {
                fmt-left    (0),
                fmt-center  (1),
                fmt-right   (2)
              } OPTIONAL,
    format-border       [8] IMPLICIT BIT STRING {
                border-left      (0),
                border-noleft    (1),
                border-top       (2),
                border-notop     (3),
                border-right     (4),
                border-noright   (5),
                border-bottom    (6),
                border-nobottom  (7)
              } OPTIONAL
            }
```

Figure C–21 illustrates the syntax used to create a language preference index construct.

Refer to the description of the corresponding DTIF$_FMI aggregate in Chapter 5.

**Figure C–21: Language Preference Index Syntax Diagram**

```
LangPrefIndex ::= INTEGER -- into LangPrefTable
```

Figure C–22 illustrates the syntax used to create a format type construct.

Refer to the description of the corresponding DTIF$_FMI aggregate in Chapter 5.

**Figure C–22: Format Type Syntax Diagram**

```
FormatType   ::= CHOICE {
    format-numeric [0] IMPLICIT SEQUENCE {
         numdatatype      [0] IMPLICIT BIT STRING
                                { numtyp-all      (0),
                                  numtyp-integer (1),
                                  numtyp-float    (2)
                                } DEFAULT { num-all },
         numfmt           [1] CHOICE {
                                num-std-fmt     [0] IMPLICIT NumericFmt,
                                num-editstr     [1] EditString,
                                num-editstr-id [2] IMPLICIT EditStrIndex
                                     },
         numrndtrunc      [2] IMPLICIT INTEGER
                                { round-display    (0),
                                  truncate-display (1)
                                } OPTIONAL
                          },
    format-text     [1] CHOICE {
         textfmt        [0] IMPLICIT TextFmt,
         textestr       [1] EditString,
         textestrid     [2] IMPLICIT EditStrIndex
                       },
    format-date     [2] CHOICE {
         datefmt        [0] IMPLICIT DateFmt,
         dateestr       [1] EditString,
         dateestrid     [2] IMPLICIT EditStrIndex
                      }
               }
```

Figure C–23 illustrates the syntax used to create an edit string index construct.

Refer to the description of the corresponding DTIF$_FMI aggregate in Chapter 5.

**Figure C–23: Edit String Index Syntax Diagram**

```
EditStrIndex ::= INTEGER  -- index into pref-editstrs
```

Figure C–24 illustrates the syntax used to create a numeric format type construct.

Refer to the description of the corresponding DTIF$_FMI aggregate in Chapter 5.

**Figure C–24: Numeric Format Type Syntax Diagram**

```
NumericFmt ::= SEQUENCE {
    numfmttype  [0] IMPLICIT INTEGER {
                              numfmt-general     (0),
                              numfmt-integer     (1),
                              numfmt-fixedpt     (2),
                              numfmt-scientific  (3),
                              numfmt-money       (4),
                              numfmt-comma       (5),
                              numfmt-percent     (6),
                              numfmt-phone       (7),
                              numfmt-bar         (8),
                              numfmt-text        (9)
                              } OPTIONAL,
    numfmtprec  [1] IMPLICIT FmtPrec OPTIONAL
                }
```

Figure C–25 illustrates the syntax used to create a numeric format precision syntax diagram.

Refer to the description of the corresponding DTIF$_FMI aggregate in Chapter 5.

### Figure C–25: Numeric Format Precision Syntax Diagram

```
FmtPrec    ::= SEQUENCE {
    fmtprecdigits  [0] IMPLICIT INTEGER OPTIONAL,
    fmtprecfrac    [1] IMPLICIT INTEGER OPTIONAL
                   }
```

Figure C–26 illustrates the syntax used to create a predefined text type construct. Refer to the description of the corresponding DTIF$_FMI aggregate in Chapter 5.

### Figure C–26: Predefined Text Types Syntax Diagram

```
TextFmt      ::= SEQUENCE {
    textfmttype [0] IMPLICIT INTEGER {
                                text-phone    (0),
                                text-text     (1),
                                text-repeat   (2)
                                } OPTIONAL
                }
```

Figure C–27 illustrates the syntax used to create a predefined date type construct. Refer to the description of the corresponding DTIF$_FMI aggregate in Chapter 5.

### Figure C–27: Predefined Date Types Syntax Diagram

```
DateFmt     ::= SEQUENCE {
    datefmttype  [0] IMPLICIT INTEGER {
                                date-dateonly      (0),
                                date-timeonly      (1),
                                date-dateandtime   (2)
                                } OPTIONAL,
    datefmtorder  [1] IMPLICIT INTEGER {
                                dateorder-mdy (0),
                                dateorder-dmy (1)
                                } OPTIONAL
                }
```

Figure C–28 illustrates the syntax used to create a format flags construct.

Refer to the description of the corresponding DTIF$_FMI aggregate in Chapter 5.

**Figure C–28: Format Flags Syntax Diagram**

```
FmtFlags  ::= BIT STRING {
  fmt-readonly          (0),
  fmt-noreadonly        (1),
  fmt-bold              (2),
  fmt-nobold            (3),
  fmt-italic            (4),
  fmt-noitalic          (5),
  fmt-underline         (6),
  fmt-nounderline       (7),
  fmt-valuehidden       (8),
  fmt-novaluehidden     (9),
  fmt-formulahidden    (10),
  fmt-noformulahidden  (11),
  fmt-running          (12),
  fmt-norunning        (13)
                 }
```

Figure C–29 illustrates the syntax used to create a date time construct.

Refer to the description of the corresponding DTIF$_DAT aggregate in Chapter 5, to the CFE$_DAT aggregate and CFE$_EXL aggregate in Chapter 6 and to the ESF$_DAT aggregate in Chapter 7.

**Figure C–29: Date Time Syntax Diagram**

```
DateTime ::= SEQUENCE {
     datetime   [0] IMPLICIT OCTET STRING,
     time-diff  [1] CHOICE {
          UTC-time   [0] IMPLICIT NULL,
          plus-diff  [1] IMPLICIT OCTET STRING,
          neg-diff   [2] IMPLICIT OCTET STRING
                         } OPTIONAL
                     }
```

Figure C–30 illustrates the syntax used to create a date time construct.

**Figure C–30: Application Private Syntax Diagram**

```
ApplPrivate ::= NamedValueList
```

Figure C–31 illustrates the syntax used to create a named value list construct.

Refer to the description of the corresponding DTIF$_NVL aggregate in Chapter 5 and to the ESF$_NVL aggregate in Chapter 7.

**Figure C–31: Named Value List Syntax Diagram**

```
NamedValueList ::= SEQUENCE OF NamedValue

NamedValue ::= SEQUENCE {
    value-name  NamedValueTag,
    value-data  ValueData
                }
```

Figure C–32 illustrates the syntax used to create a value data construct.

Refer to the description of the corresponding DTIF$_NVL aggregate in Chapter 5.

**Figure C–32: 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-list       [5] IMPLICIT SEQUENCE OF ValueData,
          value-external   [6] IMPLICIT EXTERNAL,
          value-float      [7] IMPLICIT FLOATING-POINT,
          value-date       [8] IMPLICIT DateTime,
          value-expr       [9] Expression
                     }
```

Figure C–33 illustrates the syntax used to create an ASCII string construct.

Refer to the description of the corresponding DTIF$_CAT, DTIF$_DSC, DTIF$_ERF, DTIF$_HDR, and DTIF$_NES aggregates in Chapter 5.

**Figure C–33: ASCII String Syntax Diagram**

```
ASCIIString ::= Latin1-String    --- limited to ASCII character set
```

Figure C–34 illustrates the syntax used to create a column number construct.

Refer to the description of the corresponding DTIF$_CCD aggregate in Chapter 5 and to the CFE$_EXL aggregate in Chapter 6.

**Figure C–34: Column Number Syntax Diagram**

```
ColNum  ::= INTEGER
```

Figure C–35 illustrates the syntax used to create a row number construct.

Refer to the description of the corresponding DTIF$_CCD aggregate in Chapter 5 and to the CFE$_EXL aggregate in Chapter 6.

**Figure C–35: Row Number Syntax Diagram**

```
RowNum  ::= INTEGER
```

Figure C–36 illustrates the syntax used to create a cell coordinates construct.

Refer to the description of the corresponding DTIF$_CCD aggregate in Chapter 5 and to the CFE$_CCD aggregate and CFE$_EXL aggregate in Chapter 6.

**Figure C–36: Cell Coordinates Syntax Diagram**

```
CellCoord ::= SEQUENCE {
    cell-row    [0] IMPLICIT RowNum,
    cell-column [1] IMPLICIT ColNum,
    cell-flags  [2] IMPLICIT INTEGER {
                      relrow-relcol (0),
                      relrow-abscol (1),
                      absrow-relcol (2),
                      absrow-abscol (3)
                      } DEFAULT {relrow-relcol}
                }
```

Figure C–37 illustrates the syntax used to create a range definition construct.

Refer to the description of the corresponding DTIF$_RNG aggregate in Chapter 5.

### Figure C–37: Range Definition Syntax Diagram

```
RangeDefnList ::= SEQUENCE OF RangeDefn

RangeDefn ::= SEQUENCE {
    range-name          [0] IMPLICIT Text-String OPTIONAL,
    range-type          [1] IMPLICIT INTEGER {
                                rt-named-range  (0),
                                rt-view-range   (1),
                                rt-col-title    (2),
                                rt-row-title    (3),
                                rt-display-data (4),
                                rt-data-range   (5),
                                rt-sort-range   (6)
                                } DEFAULT {rt-named-range},
    range-region        [2] IMPLICIT SEQUENCE OF Range OPTIONAL,
    range-sort-keynum   [3] IMPLICIT INTEGER OPTIONAL
                        }
```

Figure C–38 illustrates the syntax used to create a range construct.

Refer to the description of the corresponding DTIF$_RNG aggregate in Chapter 5.

### Figure C–38: Range Syntax Diagram

```
Range ::= CHOICE {
    cell-range  [0] IMPLICIT CellRange,
    row-range   [1] IMPLICIT RowRange,
    col-range   [2] IMPLICIT ColRange,
    named-range [3] IMPLICIT NamedRange
             }
```

Figure C–39 illustrates the syntax used to create a cell range construct.

Refer to the description of the corresponding DTIF$_CLR aggregate in Chapter 5 and to the CFE$_EXL aggregate in Chapter 6.

**Figure C–39: Cell Range Syntax Diagram**

```
CellRange ::= SEQUENCE {
    range-begin [0] IMPLICIT CellCoord,
    range-end   [1] IMPLICIT CellCoord OPTIONAL
    }
```

Figure C–40 illustrates the syntax used to create a row range construct.

Refer to the description of the corresponding DTIF$_RWR aggregate in Chapter 5 and to the CFE$_EXL aggregate and CFE$_RWR aggregate in Chapter 6.

**Figure C–40: Row Range Syntax Diagram**

```
RowRange ::= SEQUENCE {
    row-begin [0] IMPLICIT RowNum,
    row-end   [1] IMPLICIT RowNum OPTIONAL
    }
```

Figure C–41 illustrates the syntax used to create a column range construct.

Refer to the description of the corresponding DTIF$_COR aggregate in Chapter 5 and to the CFE$_COR aggregate and CFE$_EXL aggregate in Chapter 6.

**Figure C–41: Column Range Syntax Diagram**

```
ColRange ::= SEQUENCE {
    col-begin [0] IMPLICIT ColNum,
    col-end   [1] IMPLICIT ColNum OPTIONAL
    }
```

Figure C–42 illustrates the syntax used to create a named range construct.

Refer to the description of the corresponding DTIF$_NMR aggregate in Chapter 5 and to the CFE$_EXL aggregate in Chapter 6.

**Figure C–42: Named Range Syntax Diagram**

```
NamedRange ::= Text-String
```

# Appendix D

# CFE Syntax Diagrams

This appendix lists the syntax diagrams for each construct defined by CFE (Canonical Form Expressions). The diagram for each construct is listed alphabetically under Syntax diagrams in the index. For example, Figure D–1 shows the syntax used to create a CFE private expression and is listed as PrivateFuncExpr under Syntax diagrams in the index. For a description of the DDIS types referred to in the syntax diagrams, see Appendix B.

Figure D–1 illustrates the syntax used to create a private function expression construct.

Refer to the description of the corresponding CFE$_EXL aggregate and to the CFE$_PFE aggregate in Chapter 6.

**Figure D–1: Private Function Expression Syntax Diagram**

```
PrivateFuncExpr ::= SEQUENCE {
      pf-facility            [0] IMPLICIT ASCIIString,
      pf-name                [1] IMPLICIT ASCIIString,
      pf-reference-label     [2] IMPLICIT ASCIIString OPTIONAL,
      pf-reference-label-type[3] IMPLICIT StorageSystemTag OPTIONAL,
      pf-return-type         [4] IMPLICIT BIT STRING
          { fncret-numeric (0), --- returns numeric value
            fncret-boolean (1), --- returns boolean value
            fncret-date    (2), --- returns date value
            fncret-text    (3)  --- returns text value
          } OPTIONAL,
      pf-params              [5] IMPLICIT SEQUENCE OF NamedParameter OPTIONAL
                             }
```

Figure D–2 illustrates the syntax used to create a storage system tag construct.

Refer to the description of the corresponding CFE$_PFE aggregate in Chapter 6.

**Figure D–2: Storage System Tag Syntax Diagram**

```
StorageSystemTag  ::= ASCIIString
```

Figure D–3 illustrates the syntax used to create a named parameter construct.

Refer to the description of the corresponding CFE$_NPM aggregate in Chapter 6.

**Figure D–3: Named Parameter Syntax Diagram**

```
NamedParameter ::= SEQUENCE {
   param-name  [0] IMPLICIT Latin1-String OPTIONAL,
   param-value [1] IMPLICIT ExpressionList
                            }
```

Figure D–4 illustrates the syntax used to create an expression construct.

Refer to the description of the corresponding CFE$_EXP aggregate in Chapter 6.

**Figure D–4: Expression Syntax Diagram**

```
Expression ::= [PRIVATE 16376] IMPLICIT SEQUENCE
        { major-version [0] IMPLICIT INTEGER OPTIONAL, - omit within DTIF
          minor-version [1] IMPLICIT INTEGER OPTIONAL, - omit within DTIF
          expr-list     [2] IMPLICIT ExpressionList
        }
```

Figure D–5 illustrates the syntax used to create an expression list construct.

Refer to the description of the corresponding CFE$_CCD aggregate, CFE$_CFT aggregate, CFE$_CLR aggregate, CFE$_COR aggregate, CFE$_DAT aggregate, CFE$_EXL aggregate, CFE$_FRF aggregate, CFE$_PEX aggregate, CFE$_PFE aggregate, CFE$_RWR aggregate, CFE$_SLL aggregate, CFE$_STF aggregate, CFE$_STP aggregate, CFE$_TXC aggregate, and CFE$_VTX aggregate in Chapter 6.

**Figure D–5: Expression List Syntax Diagram**

```
ExpressionList ::= SEQUENCE OF ExprChoice

ExprChoice ::= CHOICE
    {
    --
    -- Variables/Literals
    --
    lit-integer         [0] IMPLICIT INTEGER,
    lit-float           [1] IMPLICIT FLOATING-POINT,
    lit-text            [2] Text,
    lit-date            [3] IMPLICIT DateTime,
    lit-scaled-integer  [4] IMPLICIT SCALED-INTEGER,
    lit-complex-float   [5] IMPLICIT ComplexFloat,
    lit-vtext           [6] IMPLICIT VaryingText,  -- varying length text

    cell-coord          [8] IMPLICIT CellCoord,    -- cell coordinate
    cell-range          [9] IMPLICIT CellRange,    -- cell range
    row-range           [10] IMPLICIT RowRange,    -- row range
    col-range           [11] IMPLICIT ColRange,    -- column range
    named-range         [12] IMPLICIT NamedRange,  -- named range
    col-num             [13] IMPLICIT ColNum,      -- column number
    row-num             [14] IMPLICIT RowNum,      -- column name
    col-name            [15] IMPLICIT ASCIIString,
    current-value       [179] IMPLICIT NULL,
    identifier          [180] Text,
    --
    -- Functions:  Listed in order of decreasing (perceived) frequency
    --
    -- Arithmetic Functions
    negate              [16] IMPLICIT ExpressionList,
    add                 [17] IMPLICIT ExpressionList,
    subtract            [18] IMPLICIT ExpressionList,
    divide              [19] IMPLICIT ExpressionList,
    multiply            [20] IMPLICIT ExpressionList,
    power               [21] IMPLICIT ExpressionList,
    unary-plus         [171] IMPLICIT ExpressionList,
    percent            [172] IMPLICIT ExpressionList,
```

(continued on next page)

**Figure D–5 (Cont.): Expression List Syntax Diagram**

```
-- Boolean, Relational Expressions
if-then-else        [23] IMPLICIT ExpressionList,
not                 [24] IMPLICIT ExpressionList,  -- logical NOT
and                 [25] IMPLICIT ExpressionList,  -- logical AND
or                  [26] IMPLICIT ExpressionList,  -- logical OR
eql                 [27] IMPLICIT ExpressionList,
gtr                 [28] IMPLICIT ExpressionList,  -- greater than
geq                 [29] IMPLICIT ExpressionList,  -- greater than or equal to
lss                 [30] IMPLICIT ExpressionList,  -- less than
leq                 [31] IMPLICIT ExpressionList,  -- less than or equal to
neq                 [32] IMPLICIT ExpressionList,  -- not equal to
between             [33] IMPLICIT ExpressionList,
abs-value           [34] IMPLICIT ExpressionList,  -- absolute value
modulo              [35] IMPLICIT ExpressionList,  -- modulus
sqrt                [36] IMPLICIT ExpressionList,  -- square root

-- Statistical Functions
sum                 [37] IMPLICIT SelectorList,
avg                 [38] IMPLICIT SelectorList,    -- average
count               [39] IMPLICIT SelectorList,
min                 [40] IMPLICIT SelectorList,    -- minimum
max                 [41] IMPLICIT SelectorList,    -- maximum
stdev               [42] IMPLICIT SelectorList,    -- standard deviation
var                 [43] IMPLICIT SelectorList,    -- variance

-- Conversion Functions
cvt-to-value        [44] IMPLICIT ExpressionList,  -- convert to value
round               [45] IMPLICIT ExpressionList,
truncate            [46] IMPLICIT ExpressionList,
int                 [47] IMPLICIT ExpressionList,  -- integer
decimal-string      [151] IMPLICIT DecimalString,

-- Identification Functions
iserror             [48] IMPLICIT ExpressionList,
isblank             [49] IMPLICIT ExpressionList,
isnull              [50] IMPLICIT ExpressionList,
isdate              [51] IMPLICIT ExpressionList,
isnumber            [52] IMPLICIT ExpressionList,
isstring            [53] IMPLICIT ExpressionList,
isref               [54] IMPLICIT ExpressionList,
isnot-avail        [173] IMPLICIT ExpressionList,
isnot-calc         [174] IMPLICIT ExpressionList,

-- String Functions
str-char            [55] IMPLICIT ExpressionList,  -- string character
str-code            [56] IMPLICIT ExpressionList,  -- string characater code
str-concat          [57] IMPLICIT ExpressionList,  -- string concatenate
str-extract         [58] IMPLICIT ExpressionList,  -- string extract
str-find            [59] IMPLICIT ExpressionList,  -- string find substring
str-fixed           [60] IMPLICIT ExpressionList,  -- string fixed
str-format          [61] IMPLICIT SEQUENCE         -- string format
     { source        [0] IMPLICIT ExpressionList,
       edit-string   [1] EditString

     },
```

(continued on next page)

## Figure D–5 (Cont.): Expression List Syntax Diagram

```
str-left           [62] IMPLICIT ExpressionList,   -- extract substring left
str-length         [63] IMPLICIT ExpressionList,   -- string length
str-lower          [64] IMPLICIT ExpressionList,   -- string lowercase
str-pretty         [65] IMPLICIT SEQUENCE          -- string pretty
    { string-expr   [0] IMPLICIT ExpressionList,
      pretty-flags  [1] IMPLICIT BIT STRING
           {   pretty-collapse     (0),
               pretty-compress     (1),
               pretty-lowercase    (2),
               pretty-trim         (3),
               pretty-uncomment    (4),
               pretty-upcase       (5) }
    },

str-proper         [66] IMPLICIT ExpressionList,   -- string proper
str-repeat         [67] IMPLICIT ExpressionList,   -- string repeat
str-replace        [68] IMPLICIT ExpressionList,   -- string replace
str-reverse        [69] IMPLICIT ExpressionList,   -- string reverse
str-right          [70] IMPLICIT ExpressionList,   -- extract substring right
str-trim           [71] IMPLICIT ExpressionList,   -- string trim
str-upper          [72] IMPLICIT ExpressionList,   -- string uppercase
contains           [168] IMPLICIT ExpressionList, -- contains substring
starts             [170] IMPLICIT ExpressionList, -- string starts with

-- Choose and Lookup Functions
choose             [73] IMPLICIT ExpressionList,
index              [74] IMPLICIT ExpressionList,
vlookup            [75] IMPLICIT ExpressionList,
hlookup            [76] IMPLICIT ExpressionList,
table              [77] IMPLICIT ExpressionList,
matches            [169] IMPLICIT ExpressionList,
in-table           [152] IMPLICIT ExpressionList, -- field in table

-- Date/Time Functions
name-day           [78] IMPLICIT ExpressionList,   -- date day of the week
name-month         [79] IMPLICIT ExpressionList,   -- date month name
name-daynum        [80] IMPLICIT ExpressionList,   -- day of the week
name-monthnum      [81] IMPLICIT ExpressionList,   -- month name
now                [82] IMPLICIT NULL,
today              [83] IMPLICIT NULL,
tomorrow           [84] IMPLICIT NULL,
yesterday          [85] IMPLICIT NULL,
ext-day            [86] IMPLICIT ExpressionList,   -- date/time extraction
ext-month          [87] IMPLICIT ExpressionList,   -- date/time extraction
ext-year           [88] IMPLICIT ExpressionList,   -- date/time extraction
ext-hour           [89] IMPLICIT ExpressionList,   -- date/time extraction
ext-minute         [90] IMPLICIT ExpressionList,   -- date/time extraction
ext-second         [91] IMPLICIT ExpressionList,   -- date/time extraction
diff-day           [92] IMPLICIT ExpressionList,   -- date/difference
diff-week          [93] IMPLICIT ExpressionList,   -- date/difference
diff-month         [94] IMPLICIT ExpressionList,   -- date/difference
```

(continued on next page)

## Figure D–5 (Cont.): Expression List Syntax Diagram

```
diff-year          [95] IMPLICIT ExpressionList,  -- date/difference
diff-hour          [96] IMPLICIT ExpressionList,  -- date/difference
diff-min           [97] IMPLICIT ExpressionList,  -- date/difference
diff-sec           [98] IMPLICIT ExpressionList,  -- date/difference
cvt-to-date        [99] IMPLICIT ExpressionList,  -- convert string to date
cvt-to-time        [100] IMPLICIT ExpressionList, -- convert string to time
plus-days          [101] IMPLICIT ExpressionList, -- date/time addition
plus-weeks         [102] IMPLICIT ExpressionList, -- date/time addition
plus-months        [103] IMPLICIT ExpressionList, -- date/time addition
plus-years         [104] IMPLICIT ExpressionList, -- date/time addition
plus-hours         [105] IMPLICIT ExpressionList, -- date/time addition
plus-mins          [106] IMPLICIT ExpressionList, -- date/time addition
plus-secs          [107] IMPLICIT ExpressionList, -- date/time addition

-- Cell-Related Functions
error              [108] IMPLICIT NULL,
null               [109] IMPLICIT NULL,
cur-row            [110] IMPLICIT NULL, -- current row
cur-col            [111] IMPLICIT NULL, -- current column
cur-cell           [112] IMPLICIT NULL, -- current cell
cell-row           [113] IMPLICIT ExpressionList, -- cell row
cell-col           [114] IMPLICIT ExpressionList, -- cell column
cell-name          [115] IMPLICIT ExpressionList, -- cell name
count-rows         [116] IMPLICIT ExpressionList, -- count rows
count-cols         [117] IMPLICIT ExpressionList, -- count columns
cell-extract       [118] IMPLICIT ExpressionList, -- cell extract
not-avail          [175] IMPLICIT NULL, -- not available
not-calc           [176] IMPLICIT NULL, -- not calculable
cell-indirect      [177] IMPLICIT ExpressionList, -- cell indirect

-- Financial Functions
apprec             [119] IMPLICIT ExpressionList, -- appreciation
dep-cross          [120] IMPLICIT ExpressionList, -- depreciation, declining
                                                     balance w/ crossover
                                                     to straight line
dep-db             [121] IMPLICIT ExpressionList, -- depreciation, declining
                                                     balance
dep-ddb            [122] IMPLICIT ExpressionList, -- depreciation, double
                                                     declining balance
dep-sline          [123] IMPLICIT ExpressionList, -- depreciation, straight line
dep-soyd           [124] IMPLICIT ExpressionList, -- depreciation, sum of year's
                                                     digits
discount           [125] IMPLICIT ExpressionList, -- discount
fv                 [126] IMPLICIT ExpressionList, -- future value
fva                [127] IMPLICIT ExpressionList, -- future value of an annuity
fvpv               [128] IMPLICIT ExpressionList, -- future value of a single sum
interest           [129] IMPLICIT ExpressionList, -- interest payments
irr                [130] IMPLICIT ExpressionList, -- internal rate of return
mirr               [131] IMPLICIT ExpressionList, -- modified internal rate of
                                                     return
npv                [132] IMPLICIT ExpressionList, -- net present value
payback            [133] IMPLICIT ExpressionList, -- payback
perpmt             [134] IMPLICIT ExpressionList, -- number of periods to achieve
                                                     future value
perpv              [135] IMPLICIT ExpressionList, -- number of periods given
                                                     present value
```

(continued on next page)

**Figure D–5 (Cont.): Expression List Syntax Diagram**

```
pmtpv             [136] IMPLICIT ExpressionList, -- payment per period given
                                                    present value
pmtfv             [137] IMPLICIT ExpressionList, -- payment per period to achieve
                                                    future value
principal         [138] IMPLICIT ExpressionList, -- principal
pva               [139] IMPLICIT ExpressionList, -- present value of an annuity
pvfv              [140] IMPLICIT ExpressionList, -- present value to achieve
                                                    future value
rate              [141] IMPLICIT ExpressionList, -- interest rate
perfv             [178] IMPLICIT ExpressionList, -- periods to achieve future
                                                    value


-- Series Functions
logest            [142] IMPLICIT ExpressionList, -- logest
lsqr              [143] IMPLICIT ExpressionList, -- least squares
integrate         [144] IMPLICIT ExpressionList, -- integrate
sigma             [145] IMPLICIT ExpressionList, -- sigma
trend             [146] IMPLICIT ExpressionList, -- trend

-- Additional Constants
lit-true          [147] IMPLICIT NULL, -- TRUE
lit-false         [148] IMPLICIT NULL, -- FALSE
lit-pi            [149] IMPLICIT NULL, -- PI

-- Miscellaneous Functions
random-u          [150] IMPLICIT NULL,              -- random number
sign              [167] IMPLICIT ExpressionList,    -- sign
parenthesized       [7] IMPLICIT ParenthesizedExpr, -- parenthesized expression
private-function  [998] IMPLICIT PrivateFuncExpr,   -- private Function
field-reference   [999] IMPLICIT FieldRef,          -- field Reference

-- Trigonometric Functions
sin               [153] IMPLICIT ExpressionList, -- sine
cos               [154] IMPLICIT ExpressionList, -- cosine
tan               [155] IMPLICIT ExpressionList, -- tangent
asin              [156] IMPLICIT ExpressionList, -- arc sine
acos              [157] IMPLICIT ExpressionList, -- arc cosine
atan              [158] IMPLICIT ExpressionList, -- arc tangent
atan2             [159] IMPLICIT ExpressionList, -- arc tangent 2

-- Transcendental Functions
log10             [160] IMPLICIT ExpressionList, -- log, base 10
logn              [161] IMPLICIT ExpressionList, -- log, base e
alog              [162] IMPLICIT ExpressionList, -- antilog
factorial         [163] IMPLICIT ExpressionList, -- factorial
exponent           [22] IMPLICIT ExpressionList, -- exponent

-- Binary Functions
asl               [164] IMPLICIT ExpressionList, -- arithmetic shift left
asr               [165] IMPLICIT ExpressionList, -- arithmetic shift right
ones-cmp          [166] IMPLICIT ExpressionList, -- one's complement

} -- End of ExprChoice CHOICE
```

Figure D–6 illustrates the syntax used to create a text construct.

Refer to the description of the corresponding CFE$_EXL aggregate and to the CFE$_TXC aggregate in Chapter 6.

**Figure D–6: Text Syntax Diagram**

```
Text ::= CHOICE {
                 latin1-text  [0] IMPLICIT Latin1-String,
                 simple-text  [1] IMPLICIT Character-String,
                 complex-text [2] IMPLICIT Text-String
                }
```

Figure D–7 illustrates the syntax used to create a varying text construct.

Refer to the description of the corresponding CFE$_EXL aggregate and to the CFE$_VTX aggregate in Chapter 6.

**Figure D–7: Varying Text Syntax Diagram**

```
VaryingText  ::= DTIF.VaryingText
```

Figure D–8 illustrates the syntax used to create a selector list construct.

Refer to the description of the corresponding CFE$_EXL aggregate and to the CFE$_SLL aggregate in Chapter 6.

**Figure D–8: Selector List Syntax Diagram**

```
SelectorList ::= SEQUENCE
    { criteria  [0] IMPLICIT ExpressionList OPTIONAL, - defaults to TRUE
      selection [1] IMPLICIT ExpressionList
    }
```

Figure D–9 illustrates the syntax used to create a decimal string construct.

Refer to the description of the corresponding CFE$_EXL aggregate in Chapter 6.

**Figure D–9: Decimal String Syntax Diagram**

```
DecimalString  ::= Latin1-String  --- consisting of 0-9, ., +, -
```

Figure D–10 illustrates the syntax used to create an edit string construct.

Refer to the description of the corresponding CFE$_EXL aggregate and to the CFE$_STF aggregate in Chapter 6.

**Figure D–10: Edit String Syntax Diagram**

```
EditString  ::= ESF.EditString
```

Figure D–11 illustrates the syntax used to create a parenthesized expression construct.

Refer to the description of the corresponding CFE$_EXL aggregate and to the CFE$_PEX aggregate in Chapter 6.

**Figure D–11: Parenthesized Expressions Syntax Diagram**

```
ParenthesizedExpr ::= SEQUENCE {
    begin-expr  [0] IMPLICIT Text-String   OPTIONAL,
    value-expr  [1] IMPLICIT ExpressionList,
    end-expr    [2] IMPLICIT Text-String   OPTIONAL
                }
```

Figure D–12 illustrates the syntax used to create a field reference construct.

Refer to the description of the corresponding CFE$_EXL aggregate and to the CFE$_FRF aggregate in Chapter 6.

**Figure D–12: Field Reference Syntax Diagram**

```
FieldRef ::= SEQUENCE
  {
    field-context   [0] IMPLICIT ContextVariable OPTIONAL,
    field-path      [1] IMPLICIT SEQUENCE OF Latin1-String
  }
```

# Appendix E

# ESF Syntax Diagrams

This appendix lists the syntax diagrams for each construct defined by ESF (Edit String Format). The diagram for each construct is listed alphabetically under Syntax diagrams in the index. For example, Figure E–1 shows the syntax used to create an ESF edit string and is listed as EditString under Syntax diagrams in the index. For a description of the DDIS types referred to in the syntax diagrams, see Appendix B.

Figure E–1 illustrates the syntax used to create an edit string construct.

Refer to the description of the corresponding ESF$_EDS aggregate in Chapter 7.

**Figure E–1: Edit String Syntax Diagram**

```
EditString ::= [PRIVATE 16375] IMPLICIT SEQUENCE
 {major-version [0] IMPLICIT INTEGER OPTIONAL, --- omit within DTIF
  minor-version [1] IMPLICIT INTEGER OPTIONAL, --- omit within DTIF
  edit-string   [2] IMPLICIT EditStrBuff
 }
```

Figure E–2 illustrates the syntax used to create an edit string buffer construct.

Refer to the description of the corresponding ESF$_EDS aggregate in Chapter 7.

**Figure E–2: Edit String Buffer Syntax Diagram**

```
EditStrBuff  ::= SEQUENCE OF CHOICE
                     { single-tag       Single,
                       repeat-tag [0]   IMPLICIT Repeat
                     }
```

Figure E–3 illustrates the syntax used to create a single construct.

Refer to the description of the corresponding ESF$_EDS aggregate and to the ESF$_TXT aggregate in Chapter 7.

**Figure E–3: Single Syntax Diagram**

```
Single ::=  CHOICE {
         alphabetic        [1] IMPLICIT NULL,
         am-pm             [2] IMPLICIT NULL,
         any-char          [3] IMPLICIT NULL,
         any-case          [4] IMPLICIT NULL,
         binary-digit      [5] IMPLICIT NULL,
         digit-sep         [6] IMPLICIT NULL,
         day-number        [7] IMPLICIT NULL,
         decimal-digit     [8] IMPLICIT NULL,
         radix-point       [9] IMPLICIT NULL,
         encoded-minus     [10] IMPLICIT NULL,
         encoded-plus      [11] IMPLICIT NULL,
         encoded-sign      [12] IMPLICIT NULL,
         exponent          [13] IMPLICIT NULL,

         zero-replace      [14] IMPLICIT Text-String,
         currency          [15] IMPLICIT NULL,
         minus             [16] IMPLICIT NULL,
         plus              [17] IMPLICIT NULL,
         sign              [18] IMPLICIT NULL,
         float-blank-supr  [19] IMPLICIT NULL,
         fraction-second   [20] IMPLICIT NULL,
         hex-digit         [21] IMPLICIT NULL,
         hour-12           [22] IMPLICIT NULL,
         hour-24           [23] IMPLICIT NULL,
         julian-digit      [24] IMPLICIT NULL,
         logical-char      [25] IMPLICIT NULL,
         long-text         [26] IMPLICIT NULL,
         lowercase         [27] IMPLICIT NULL,
         minus-literal     [28] IMPLICIT Text-String,
         minus-lit-end     [29] IMPLICIT Text-String,
         minute            [30] IMPLICIT NULL,
         month-name        [31] IMPLICIT NULL,
         month-number      [32] IMPLICIT NULL,
         octal-digit       [33] IMPLICIT NULL,
         plus-literal      [34] IMPLICIT Text-String,
         reverse           [35] IMPLICIT NULL,
         second            [36] IMPLICIT NULL,
         str-literal       [37] IMPLICIT Text-String,
         missing-sep       [38] IMPLICIT NULL,
         uppercase         [39] IMPLICIT NULL,
         weekdayname       [40] IMPLICIT NULL,
         year              [41] IMPLICIT NULL,
         appl-private      [42] IMPLICIT ApplPrivate,
         digit-sep-lit     [43] IMPLICIT Text-String,
         radix-point-lit   [44] IMPLICIT Text-String,
         currency-lit      [45] IMPLICIT Text-String
        }
```

Figure E–4 illustrates the syntax used to create a repeat construct.

Refer to the description of the corresponding ESF$_EDS aggregate and to the ESF$_RPT aggregate in Chapter 7.

**Figure E–4: Repeat Syntax Diagram**

```
Repeat ::= SEQUENCE
    { repeat-count [0] IMPLICIT INTEGER,
      repeat-seq       Single
    }
```

Figure E–5 illustrates the syntax used to create an application private construct.

Refer to the description of the corresponding ESF$_EXT aggregate in Chapter 7.

**Figure E–5: Application Private Edit String Syntax Diagram**

```
ApplPrivate ::= NamedValueList
```

Appendix F

# 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 F.1 describes various VMS file management commands and utilities that display, create, and preserve file tags where appropriate. Section F.1 also describes the way various VMS commands and utilities respond to DDIF file input. Section F.2 describes VMS support for DDIF files in heterogeneous computing environments. Section F.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.

## F.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[1] |
| COPY | Preserves file tag[1] |
| BACKUP | Preserves file tag |

[1]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.

## F.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.

### F.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
    .
    .
    .
```

### F.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
    .
    .
    .
```

## F.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 F.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
```

## F.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 Section F.1.3.1 and Section F.1.3.2.

The Backup Utility always preserves file tags and semantics when you back up a DDIF file to magnetic tape.

### F.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
   ```

### F.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.

### F.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.

## F.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.

### F.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/TRANSFER_MODE=block/FDL=DDIF.FDL input_filespec output_filespec
```

### F.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.

### F.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.

## F.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.

### F.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 F–1.

**Table F–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 F–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 F–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),
```

(continued on next page)

```
    FAB         : $FAB( fnm = 'TAGGED-FILE.TEST',
                        nam = NAM,
                        mrs = 512,
                        rfm = FIX,
                        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 = $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
```

## F.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.

#### F.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.

#### F.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 F–2 illustrates a BLISS–32 program that accesses a tagged file from an application program that does not use an RMS extension.

**Example F–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);
```

(continued on next page)

**Example F–2 (Cont.): Accessing a Tagged File**

```
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
    SIGNAL (.STATUS);
STATUS = $CLOSE( FAB = FAB );
IF NOT .STATUS
THEN
    SIGNAL (.STATUS);
END;
END
ELUDOM
```

## F.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
```

## F.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.

## F.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.

# Appendix G

# CDA$ Facility Messages

This appendix lists the CDA$_ facility messages generated by the CDA Toolkit. A brief explanation and recommended user action follows each message, unless a user action is not required. The messages are listed in alphabetical order, by message name.

ALLOCFAIL, memory allocation failure

**Level:** Error

**Explanation:** The standard memory allocation procedure failed to allocate dynamic memory.

CLOSEFAIL, close failure

**Level:** Error

**Explanation:** The standard close function has failed.

DCVNOTFND, domain converter not found

**Level:** Error

**Explanation:** The required domain converter could not be found.

DEFAULT, item present by default

**Level:** Success

**Explanation:** The application called CDA$LOCATE_ITEM, which determined that the item was present by default in the input stream.

DEFNOTFOU, definition not found

**Level:** Error

**Explanation:** The application called CDA$FIND_DEFINITION referencing an entity that is not defined.

EMPTY, empty item

**Level:** Error

**Explanation:** The application called a CDA access procedure referencing an item that is empty.

ENDOFDOC, end of document

**Level:** Error

**Explanation:** The application called CDA$GET_AGGREGATE, which determined that no more aggregates exist in the document.

ENDOFSEQ, end of sequence

**Level:** Error

**Explanation:** The application called CDA$NEXT_AGGREGATE referencing an aggregate that was at the end of a sequence.

ERRINPLOG, error messages produced during input conversion, see error log

**Level:** Error

**Explanation:** The input conversion did not complete and some error messages were produced.

**User Action:** Refer to the error log for more details.

ERROUTLOG, error messages produced during output conversion, see error log

**Level:** Error

**Explanation:** The output conversion did not complete and some error messages were produced.

**User Action:** Refer to the error log for more details.

FLTTRN, floating-point truncation

**Level:** Error

**Explanation:** During CDA$LOCATE_ITEM for a general floating-point value, floating truncation occurred.

ICVNOTFND, input converter not found

**Level:** Error

**Explanation:** The specified input converter could not be found.

INDEX, index out of range

**Level:** Error

**Explanation:** The application called CDA$LOCATE_ITEM, CDA$STORE_ITEM, or CDA$ERASE_ITEM referencing an array-valued item, but the index is out of range.

INFINPLOG, informational messages produced during input conversion, see error log

**Level:** Informational

**Explanation:** The input conversion completed but some informational messages were produced.

**User Action:** Refer to the error log for more details.

INFOUTLOG, informational messages produced during output conversion, see error log

**Level:** Informational

**Explanation:** The output conversion completed but some informational messages were produced.

**User Action:** Refer to the error log for more details.

INHERIT, item present by inheritance

**Level:** Success

**Explanation:** The application called CDA$LOCATE_ITEM, which determined that the item was present by inheritance in the input stream.

INTERR, internal error

**Level:** Fatal

**Explanation:** The CDA Toolkit detected an internal error.

INVADDINF, invalid additional information

**Level:** Error

**Explanation:** The add-info parameter in a call to CDA$LOCATE_ITEM or CDA$STORE_ITEM is invalid.

INVAGGTYP, invalid aggregate type

**Level:** Error

**Explanation:** The application called a CDA access procedure referencing an aggregate type code that is undefined, or an aggregate that has an undefined type code.

INVBUFLEN, invalid buffer length

**Level:** Error

**Explanation:** The application called CDA$_STORE_ITEM referencing an item that is required to have a specified buffer length. The value of the buffer length parameter is not the required value.

INVDATLEN, invalid data length

**Level:** Error

**Explanation:** The length of the value data exceeded the specified length for the data type.

INVDOC, invalid document syntax

**Level:** Error

**Explanation:** The CDA access procedures determined that the document contains invalid syntax.

INVFLTVAL, invalid floating-point value

**Level:** Error

**Explanation:** A floating-point datum has a reserved value.

INVFUNCOD, invalid function code

**Level:** Error

**Explanation:** The application called CDA$CONVERT with an invalid function code.

INVINPDMN, invalid input domain

**Level:** Error

**Explanation:** An invalid input domain was specified for the front end. Only DDIF and DTIF are supported as domains.

INVINSERT, invalid insert

**Level:** Error

**Explanation:** The application called CDA$INSERT_AGGREGATE or CDA$STORE_ITEM referencing an aggregate that was already part of a sequence, but was not the first aggregate of the sequence.

INVITMCOD, invalid item code

**Level:** Error

**Explanation:** The application called a CDA access procedure referencing an aggregate item code that is not defined.

INVITMLST, invalid item list

**Level:** Error

**Explanation:** The application called CDA$OPEN_FILE or CDA$CREATE_ROOT_AGGREGATE with a processing options item list that contained an invalid item.

INVOPTION, invalid converter option

**Level:** Error

**Explanation:** An invalid option was specified for the converter.

**User Action:** Refer to the documentation for this converter to see the valid options.

INVOUTDMN, invalid output domain

**Level:** Error

**Explanation:** An invalid output domain was specified for the back end. Only DDIF and DTIF are supported as domains.

INVSCOCOD, invalid scope code

**Level:** Error

**Explanation:** The application called CDA$ENTER_SCOPE or CDA$LEAVE_SCOPE referencing a scope code that is not defined or invalid in following correct document scoping rules.

INVSCOTRAN, invalid scope transition

**Level:** Error

**Explanation:** The application made a call to CDA$ENTER_SCOPE or CDA$LEAVE_SCOPE that did not follow correct scoping rules.

INVTAGCOD, invalid tag code

**Level:** Error

**Explanation:** The application called CDA$_STORE_ITEM referencing an item that has a special tag encoding, but the value of the add-info parameter is not defined for the item.

NORMAL, normal successful completion

**Level:** Success

**Explanation:** The specified action has been successfully completed.

OCVNOTFND, output converter not found

**Level:** Error

**Explanation:** The specified output converter could not be found.

OPENFAIL, open failure

**Level:** Error

**Explanation:** The standard open function has failed.

READFAIL, read failure

**Level:** Error

**Explanation:** The standard read function has failed.

READONLY, aggregate is read-only

**Level:** Error

**Explanation:** The application requested input processing options that require an aggregate to be read-only. The application attempted to write or delete the aggregate.

SUSPEND, converter is suspended

**Level:** Success

**Explanation:** The application called CDA$CONVERT, which determined that the back end suspended conversion. This message can only be returned by the CDA viewer.

UNSUPCNV, unsupported document conversion

**Level:** Error

**Explanation:** The input and output document formats are incompatible for conversion.

UNSUPFMT, unsupported document format

**Level:** Error

**Explanation:** The application called CDA$CONVERT with an unsupported document format name. The document format name may be misspelled, or the required conversion module may not be installed.

VAREMPTY, empty variant item

**Level:** Error

**Explanation:** The application called CDA$LOCATE_ITEM, CDA$STORE_ITEM, or CDA$ERASE_ITEM referencing an item that has a variable data type. The item that specifies the data type is empty.

VARINDEX, variant index out of range

**Level:** Error

**Explanation:** The application called CDA$LOCATE_ITEM, CDA$STORE_ITEM, or CDA$ERASE_ITEM referencing an array-valued item that has a variable data type, but the index is out of range for the item that specifies the data type.

VARVALUE, variant value out of range

**Level:** Error

**Explanation:** The application called CDA$LOCATE_ITEM, CDA$STORE_ITEM, or CDA$ERASE_ITEM referencing an item that has a variable data type, but the item that specifies the data type has an invalid value.

VERSKEW, major version skew between input file and CDA Toolkit

**Level:** Error

**Explanation:** The file's major version is different from the Toolkit's. Thus, the Toolkit cannot properly process the file.

WRITFAIL, write failure

**Level:** Error

**Explanation:** The standard write function has failed.

# Glossary of Terms

This glossary alphabetically lists and defines terminology associated with the CDA architecture, data structures, and routines.

**add-info**

Additional information. This is a parameter to the LOCATE ITEM and STORE ITEM routines.

**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.

**arc**

A graphics object representing a circle or piece of a circle.

**attribute**

A term used to describe content characteristics such as font, line thickness, and color.

**back end**

A CDA application that converts a document from DDIF or DTIF format into a document in another format, such as PostScript or plain text.

**bit string**

A compound document data type that is encoded as a string of bits. The length of the item is expressed in bits.

**BMU**

Basic Measuring Unit. A BMU is 1/1200 of an inch.

**Boolean**

A compound document data type, encoded as a byte, that represents a Boolean value, which is always either True or False.

**byte**

A compound document data type that is encoded as 8-bits of storage. The term "octet" is also used to describe an 8-bit byte.

**calling format**

A calling format is the interface specification for a routine. The calling format describes how the routine is to be called by other routines. The specification includes the routine's name or address, and all of the routine's parameters.

**CDA**

Compound Document Architecture. An architecture for interchanging complex and simple data in a computer architecture independent manner.

**CFE**

Canonical Form Expressions. A CDA format for defining revisable expressions at the cell or column level within a DTIF document.

**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. For example, page numbers are often stored as computed content.

**condition value**

An error that causes normal processing to stop is called a condition. The condition value identifies the error that caused processing to stop.

**content**

The class of data that makes up the fundamental units of documents. Document content includes text, graphics, raster images, and so on. The standard content aggregates are the basic units of content.

**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.

**content tag**

CDA content is uniquely identified by the context it appears in and a tag. For example, the tag for DDIF Latin1 text content is DDIF$_TXT.

**converter**

A CDA application that converts data to or from a CDA format. See also, front end, back end, base system converter, and third-party converter.

**DDIF**

Digital Document Interchange Format. A CDA format that can be used to interchange simple or complex documents containing text, images, and/or graphics.

**DDIS**

Digital Data Interchange Syntax. A method of encoding data for machine-independent interchange. DDIS is a subset of the International Standards Organization Open Systems Interconnect ASN.1 transfer syntax.

**DTIF**

Digital Table Interchange Format. A CDA format that can be used to interchange simple or complex spreadsheet data.

**document**

An entire hierarchical structure in memory, created by the CDA Toolkit routines.

**document content**

See *content*.

**document segment**

See *segment*.

**enumeration**

Assigns names to numbers or bits. Enumeration is 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.

**ESF**

Edit String Format. A CDA format for defining revisable, user-defined edit strings that define specialized display formats for numeric, text, and date/time data values.

**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. Final form documents are generally not revisable/editable.

**floating-point**

See general floating-point and single-precision floating-point.

**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.

**frame**

A DDIF frame specifies a region in a document. The frame region may be anchored to a particular place in a document, or it may be associated with a section of text. Content within the frame is positioned relative to the frame. Graphics and image content must always be contained in a frame of the appropriate content category.

**front end converter**

A CDA application that converts a document or data from a particular format to a CDA format. For example, the WK1 front end converts Lotus 1-2-3 spreadsheets to DTIF format.

**galley**

A rectangular guide, such as a column or footnote area. DDIF galleys are modeled by areas 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.

**general floating-point**

A floating-point value for which the format is specified by the *add-in* parameter. Upon storing a floating-point value, the CDA Toolkit transforms the value to a generic DDIS H-floating-point value. During a call to the LOCATE ITEM routine, an application specifies whether the DDIS-encoded floating-point data should be converted to VAX or IEEE format by the CDA Toolkit.

**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 layout**

A set of rules that are used to determine the layout of a document or set of documents.

**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.

**GKS**

Graphical Kernel System. A standard way of specifying graphics objects.

**graphics content**

Content that consists of primitives such as arcs, 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 pixels.

**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 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.

**language**

One of the languages specified by ISO 639.

**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.

**legend**

Like the legend of a map, the DDIF legend is used to describe the coordinate system of a segment. The legend information is used to indicate the scale of an illustration. See also Measurement.

**longword**

A compound document data type representing a 32 bit-encoded structure. The bits are interpreted according to a defined structure.

**marker**

An object used to mark a place. For example, markers can be used to mark data points on a graph.

**measurement**

DDIF measurements are specified using the units-per-measure segment attribute. By default, all measurements are in BMUs, 1/1200 of an inch. Scaling can be accomplished changing the units-per-measure for a document, or by using a transformation aggregate.

**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. Object identifiers are used to uniquely identify a class of object.

**octet**

An 8-bit byte.

**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.

**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 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.

**sequence of**

A linked series of aggregates, all of which are the same type.

**single-precision floating-point**

A VAX F_floating-point value on VAX systems; an IEEE Standard 754 single-precision floating-point value on non-VAX systems. The length of the buffer is always 4. Upon writing a floating-point value to a DDIS stream, the CDA Toolkit transforms the value to a generic DDIS floating-point value. When reading a single-precision floating-point value from a DDIS stream, the DDIS-encoded floating-point data is converted to the native (VAX or IEEE) format by the CDA Toolkit.

**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.

**specific layout**

The layout of a particular document or document element.

**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 string whose character set is identified in an add-info parameter.

**style guide**

A relationship in a revisable document that defines a collection of generic types defined for use by a set of documents. For example, a style guide could be used to define a default newspaper layout or a standard letterhead.

**tag**

Tags are used to identify the type of a CDA item. For example, $2D is used to identify a segment's content category as graphics.

**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.

**Viewer**

The CDA Viewer is an application that can be used to look at a CDA (DDIF or DTIF) format document on a character-cell or a DECwindows display.

**widget**

A DECwindows object providing a user-interface abstraction, for example, a scroll bar or the CDA Viewer.

**word**

A compound document data type that is encoded as a 16-bit structure.

**wrapping**

The process of breaking a stream of characters into lines that fit within the assigned galleys.

# Index

## A

# B



# C

## D

# E

# F

## G

## H



## I

# K



# L

# M



# N

# O



# P

# R

# S

# T

## U



## V

## W



## X



## Y

# 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<br>P.O. Box CS2008<br>Nashua, New Hampshire 03061 |
| Puerto Rico | 809-754-7575 | Local Digital subsidiary |
| Canada | 800-267-6215 | Digital Equipment of Canada<br>Attn: DECdirect Operations KAO2/2<br>P.O. Box 13000<br>100 Herzberg Road<br>Kanata, Ontario, Canada K2K 2A6 |
| International | ———— | Local Digital subsidiary or approved distributor |
| Internal[1] | ———— | USASSB Order Processing - WMO/E15<br>*or*<br>U.S. Area Software Supply Business<br>Digital Equipment Corporation<br>Westminster, Massachusetts 01473 |

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

# Reader's Comments

**CDA Reference Manual**
**Volume 2**
AA–PC02A–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

**CDA Reference Manual**
**Volume 2**
AA–PC02A–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