# E C M A

EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION

# STANDARD ECMA - 187

## OFFICE DOCUMENT ARCHITECTURE (ODA)
## -
## APPLICATION PROGRAMMING INTERFACE
## -
## APPLICATION PROFILE INTERFACE FOR HANDLING COMPOUND DOCUMENTS

### Volume 1



June 1993

Free copies of this document are available from ECMA,
European Computer Manufacturers Association,
114 Rue du Rhône - CH-1204 Geneva (Switzerland)

Phone: +41 22 735 36 34 Fax: +41 22 786 52 31
X.400: C=ch, A=arcom, P=ecma, O=genevanet,
OU1=ecma, S=helpdesk
Internet: helpdesk@ecma.ch

# E C M A

EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION

# STANDARD ECMA - 187

## OFFICE DOCUMENT ARCHITECTURE (ODA)
## -
## APPLICATION PROGRAMMING INTERFACE
## -
## APPLICATION PROFILE INTERFACE FOR HANDLING COMPOUND DOCUMENTS

### Volume 1



June 1993

## Brief History

In 1985, ECMA/TC29 published Standard ECMA-101, Office Document Architecture, in order to facilitate the interchange of documents.

In the meantime, work had started in ISO and CCITT, resulting in the preparation of ISO 8613 "Office Document Architecture (ODA) and interchange format". Experts of TC29 participated to the work, and acted as editors of most of the parts of the ISO and CCITT documents.

The second edition of Standard ECMA-101 was prepared in 1988 by ECMA/TC29 in order to align ECMA-101 with the last ISO and CCITT publications (ISO 8613 and CCITT Recommendations in the T.410 series).

Work started in 1991 in a group of ECMA/TC29 to prepare application programming interfaces for use with Standard ECMA-101. Two API specifications were agreed for handling compound documents, one at constituent level and one at application profile level (this Standard). Both specifications are aligned with the current ECMA, ISO and CCITT standards, and with the ODA Document Application Profiles approved by ISO (FOD11, FOD26 and FOD36). Representatives of ISO/IEC JTC1/SC18 attended the meetings of ECMA-TC29.

---

---

This ECMA Standard is divided in three volumes, as follows:

**Volume 1**

1 General
2 Introduction
3 References
  3.1 Data Types
  3.2 Constants
  3.3 Error Handling

**Volume 2**

  3.4 Functions

**Volume 3**

Annexes

This is volume 1.

Adopted as an ECMA Standard by the General Assembly of June 1993

## Table of Contents

# 1 Introduction

This Standard provides reference material for using the Document Application Profile (DAP) level Application Programming Interface (API), to create applications and services conforming to the Draft International Standardized Profiles (DISPs) Format Open Documents 11, 26 and 36 (FODs 11, 26 and 36).

This Standard is intended for system and application programmers who are familiar with FOD11, FOD26, FOD36 and Open Document Architecture (ODA).

## 1.1 Structure of this Standard

This Standard is structured as follows:

- clause 1 (this clause) contains introductory material;
- clause 2 introduces the DAP Level API;
- clause 3 gives reference material and describes the functions provided by the DAP Level API;
- annex A lists and classifies the properties; it gives the data types and further information about the properties;
- annex B lists the properties for each entity type;
- annex C provides a clarification of certain aspects of FOD11 and FOD26;
- annex D provides a clarification of certain aspects of FOD36;
- annex E, Glossary defines terms used in this Standard.

Error messages for each function are listed and explained under the relevant function section in clause 3.

## 1.2 References

Associated documents include:

| | |
|---|---|
| *ISO 8613(1989):* | *Information Processing: Text and Office Document Architecture and Interchange Format.* |
| *ISO 8613* | *Technical Corrigenda.* |
| *FOD11* | *Draft International Standardized Profile (ISO/IEC DISP 10610-1* |
| *FOD26* | *Draft International Standardized Profile (ISO/IEC DISP 11181-1)* |
| *FOD36* | *Draft International Standardized Profile (ISO/IEC DISP 11182-1)* |

## 3.3 Conventions

The following table shows the conventions used in this Standard.

| Convention | Description |
|---|---|
| *Italics* | Italics are used for new terms and to refer to other documents. |
| `Monospaced font` | Monospaced font is used for code fragments and examples. |
| UPPER CASE | Upper case is used for property codes, constants and error codes. |
| *Note:* | Notes contain information of special importance to the reader. |

# 2 Introduction to the DAP Level API

The Document Application Profile (DAP) Level API provides an Application Programming Interface (API) for a set of services. This set of services supports the creation and reading of Open Document Architecture (ODA) documents at the level of the Draft International Standardized Profiles (DISPs) Format Open Documents 11, 26 and 36 (FOD11, FOD26 and FOD36).

This clause provides an introduction to the reference material for the DAP Level API. It covers the following areas:

- DAP Level API concepts
  - DAP types, entities and properties
  - Initialization and termination
  - Creating documents
  - Reading DAP-conformant documents
  - DAP API details
- Writing applications
- Restrictions
- Dependencies

## 2.1 Concepts

The DAP Level API Tool provides an API for common services supporting the generation and interpretation of ODA documents according to the constraints defined by the FOD11, FOD26 and FOD36 DISPs.

To simplify application development, the interface operates in terms of DAP features, such as footnote and numbering style, rather than in terms of the ODA constituents used to represent them.

The DAP Level API Tool does not generate all possible conformant data streams. It generates a useful subset of these data streams, with the restrictions that are identified in this manual. The DAP Level API Tool accepts any correctly conformant document, but interprets it only within the given restrictions. For further information about restrictions, see 2.3 and Appendices C and D.

The DAP Level API Tool depends on the ODA Level API Tool. It does not depend on any of the other Tools in the Toolkit.

Using the DAP Level API to create an ODA document aids conformance to the appropriate DAP, provided that the same document is not manipulated directly via the ODA Level API.

The DAP Level API Tool provides a single API; it does not provide separate APIs for each DISP supported. The API includes functions to define the DAP level for each new document that is created, independently of other documents being handled at the same time. Once the DAP level of any one document is established, the level can only be changed by using the ODA Level API. The DAP Level API can then play no further part in aiding conformance to this new level.

The DAP API contains appropriate functionality for various services. These services are given in the following list, using ODA terminology:

- Management of all the constituent constraints defined by the DAP for the following ODA constituents:
  - Objects that correspond to the specific logical objects or generic logical object classes
  - Objects that correspond to the generic layout objects, which define the page format, headers and footers etc.
  - Objects that correspond to the specific layout objects, for read access only
  - Layout styles and presentation styles
  - Document profile
  - Content portions
- Interface and document naming that is consistent with DISP terminology.
- Support of operations in terms of DAP features, such as footnote, that relate to several ODA constituents.

- The user option to infer generic logical object classes from the specific logical objects and the DAP rules. Information that is necessary to build the logical classes is passed through the objects corresponding to the specific logical objects, for example:
  - Automatic numbering features
  - Generic layout styles and generic presentation styles
- Implicit use of the Bindings attribute and the Content generator attribute in order to support content generation, such as automatic numbering.
- User selectable factorization of logical classes, layout styles, and presentation styles.
- Management of any computer memory used by the DAP API Tool so that the usage is transparent to any application.

The DAP API provides functions to support the creation and reading of DAP-conformant documents. The DAP API does not support the updating of an existing document.

The DAP API provides a higher level interface for applications than that provided by the ODA API. It enables the application to operate in terms of DAP entities and DAP properties rather than in terms of ODA constituents and their attributes. (See 2.1.1 for information about DAP entities and properties.) As far as possible, this provides a consistent architectural view of the document and rationalizes the conventions for using ODA as prescribed by the DAP constraints. This view of documents hides the underlying ODA architecture such that, depending on its mode of access to the document, an application can be isolated from any unwanted detail and from the complexity of ODA.

The DAP API is a self-contained stand-alone interface, that must not be bypassed. This means that ODA API calls and DAP API calls must not be mixed on the same document instance. Neither API prevents such misuse, but the results are undefined.

The DAP API provides two primary modes of operation for writing documents: mode-1 and mode-2. Mode-1 provides for the automatic creation of a generic logical structure while the specific logical structure is created explicitly. Mode-2 provides for and requires the explicit creation by the application of the generic logical structure, and offers full control of the population and disposition of presentation and layout styles. If mode-2 is established when the document is created, it cannot subsequently be changed.

If a document is created referring to an external document class (generic document) then a separate mode must be used. See 2.1.3.1 for information about the requirement for separate modes.

### 2.1.1 DAP Types, Entities and Properties

A DAP-conformant document is considered to consist entirely of a related set of *DAP entities*. A DAP entity is a set of *DAP properties*. An entity is of a particular *DAP type*. Explanations of these terms are given in this clause.

*DAP types* generally correspond directly to the document profile constituent constraints, logical constituent constraints, and layout constituent constraints defined in clause 7 of an International Standards Organization (ISO) ISP. DAP types are identified by the same names as those constituent constraints. The DAP types corresponding to layout style constraints and presentation style constraints have a limited independence. This is because for most purposes they are read and written as adjuncts to other entities. The different types of styles identified in the DAPs, but not identified as such in datastreams, are only distinguished by the DAP API by virtue of the entities to which they are attached. There are no DAP types corresponding to content portion constraints.

*DAP entities* can be specific or generic. They are generally referred to by their DAP type. For example, a specific DAP entity of DAP type *paragraph* is called a *specific paragraph*. When writing documents, styles can also be thought of as specific or generic because each one can only be used in either specific logical structure or generic logical structure, but not in both.

Wherever it is feasible, the DAP API defines a set of related DAP entities as a *cluster*. A cluster is a hierarchy of DAP entities within which the FOD has defined that the subordinates are mandatory. The use of clusters reduces work for the application, because functions such as dla_create_specific_entity and dla_duplicate_specific_entity can operate on a cluster as a whole by addressing the most superior DAP entity in the hierarchy. FOD36 defines additional structures that are similar to clusters, but subject to more complex

constraints. These structures are called *clumps*. The main difference between a cluster and a clump is as follows: in a cluster all the entities are mandatory; in a clump the application has a certain amount of choice as to which entities should be used to make up the clump, but once this choice has been made the selected entities then become mandatory and the clump behaves as a cluster. Further information about clumps is given in 2.2.2.4.

A *DAP property* is an element of a DAP entity that has a name and a type of value defined at the DAP API. DAP properties of data type SEQUENCE, CHOICE or nonvalued may be named groups of subsidiary DAP properties. SEQUENCE is used to contain an indexed repetition of another DAP property. CHOICE is used when the value can be a choice of one of several different properties. Nonvalued is used to refer to a group of related properties. Operations on DAP properties that are groups of other properties are limited to setting the status to null, or getting information about the group. This information refers to size in the case of SEQUENCE, and to which subsidiary property in the case of CHOICE.

A DAP property expresses one characteristic of the DAP entity or a relationship with one or more DAP entities. The property can correspond to an ODA attribute, parameter, or subparameter. DAP properties include those corresponding to ODA attributes that are specified for presentation styles and layout styles.

Styles have a limited visibility at the interface. When writing documents, the DAP API creates and extends styles automatically, sufficient to represent relevant DAP properties set on DAP entities. The only DAP properties that can be directly read (and in some circumstances written) for styles are User visible name, User readable comments and Identifier.

There is generally a correspondence between DAP properties and ODA attributes that underlie them. However, some ODA attributes are not directly represented at the DAP API. This is usually because the FOD mandates a single value for the property; it can therefore be set automatically by the DAP API and is of no interest when reading a document. In other cases, possibilities inherent in a single attribute through the use of different values are represented, for clarity, by distinct properties.

ODA attributes involving expressions, such as Bindings, Content generators and Generator for subordinates are not directly represented at the DAP API. Where relevant to the application, the semantic content of the expressions permitted within the DAP constraints is represented at the DAP API by simple integer and string valued properties. No special functions are required to manipulate the properties corresponding to these complex attributes.

Generally, the application can get and set the values of DAP properties, especially DAP properties of type string, integer (where integer includes predefined names as well as numeric values) and entity handle. See annex A for information about property types.

Certain DAP property values, such as entity type, are set implicitly as a by-product of a function, but the application can get the values explicitly. Some DAP properties, for example Tab stop or Character sets, have several values that are accessed as arguments of special functions for those DAP properties.

### 2.1.2 Initialization and Termination

An instance of the DAP API Tool must be initialized before carrying out any document processing, and terminated when the document processing is complete. The DAP API provides functions to:

- Create a new DAP document or to commence the reading of an existing DAP document.
- Enable a document to refer to an external document class or resource document.
- Write a copy of a new document as an Open Document Interchange Format (ODIF) data stream. ODIF data stream constructs such as ODA identifiers are automatically completed.
- Delete a DAP document in order to relinquish the resources that it occupies.
- Register a callback function to be called periodically by the DAP API during the reading or writing of documents to or from ODIF. This is so that the progress of lengthy operations can be monitored, or the processor can be yielded in non-preemptive operating systems.

### 2.1.3 Creating Documents

Only processable form documents can be created; these can refer to an external document class or to a resource document. Documents intended for use as external document class or resource documents can also be created.

When a document is created it must be given a DAP level and a mode of operation. The DAP level and mode constrain the functions that can be performed on the particular document, in order to assist conformance to the stipulated DAP level.

The DAP level given to a document cannot be changed, and only one particular change in mode is permitted. See 2.1.3.1 for further information about modes and changing modes.

The DAP API supports functions (depending on the mode) to:

- Create a new document and establish the writing mode. A document profile and, in mode-1, document root(s) are automatically created.
- Establish a link with a pre-existing external document class or resource document.
- Create and link DAP entities corresponding to a complete generic logical structure.
- Create and link DAP entities corresponding to a complete generic layout structure.
- Create and link DAP entities corresponding to the specific logical structure. In mode-1 a corresponding complete generic logical structure is automatically created.
- Set DAP properties on the newly created DAP entities.
- Assign automatically created styles to other DAP Entities.
- Add content information to relevant DAP entities.

### 2.1.3.1 Modes of Operation

When a document is created it must be given a mode of operation. The constant values that represent these modes are:

```
DLA_C_NO_OPTIONS (= DLA_C_AUTO_GEN)
DLA_C_MODE_1 (= DLA_C_NO_OPTIONS)
DLA_C_AUTO_GEN (= DLA_C_NO_OPTIONS)
DLA_C_MODE_2
DLA_C_WITH_EXTERNAL
DLA_C_FOR_EXTERNAL
DLA_C_FOR_RESOURCE
```

The first three constant values (DLA_C_NO_OPTIONS, DLA_C_MODE_1 and DLA_C_AUTO_GEN) are equivalent, and this mode (mode-1) provides for the automatic creation of a generic logical structure while a specific logical structure is created explicitly. The entities within this generic logical structure can be read but cannot be altered unless the mode is changed to DLA_C_MODE_2. This is the only mode alteration that is allowed. Mode-1 requires an application to construct a generic layout structure explicitly, using functions provided for that purpose.

The next distinct mode is mode-2 (DLA_C_MODE_2), which permits and requires an application to construct both the generic logical and the generic layout structures explicitly, using functions provided for that purpose. Each generic entity, and all links between generic entities, (corresponding to the ODA attribute Generator for subordinates) must be established by the application. Each entity in the specific logical structure is created as an instance of an entity that has been created previously in the generic logical structure. This is the only writing mode in which a generic document can be read as a resource document for the specific document.

The mode DLA_C_WITH_EXTERNAL provides for the use of an external document class generic document. Use of this mode prohibits the creation of either of the generic structures because this would invalidate the external document class.

The modes DLA_C_FOR_EXTERNAL and DLA_C_FOR_RESOURCE are provided so that documents having no specific structures can be created for subsequent use as external or resource documents. In both of these modes the application must supply the ODIF identifiers for all entities, including styles. The difference between these two modes is that an external document must match the conformance rules for the generic part of a complete document at the given DAP level, whereas a resource document has entirely different conformance rules. A resource document created by the DAP API is restricted to contain only the prescribed

entities as indicated by the FODs; it cannot have generic roots and is not created in the same top-down manner as all other documents.

#### 2.1.3.2 Creating and Linking DAP Entities

The DAP API provides functions to create a new specific or generic DAP entity as an instance of a DAP type, and to create a specific DAP entity as a copy of an existing specific DAP entity. In mode-2 and in mode DLA_C_WITH_EXTERNAL there is also a function to create a specific DAP entity as an instance of a generic DAP entity.

When a new DAP entity is created, it is always linked to an existing DAP entity as a subordinate of that existing entity. The existing entity can be the appropriate document root. (The mode DLA_C_FOR_RESOURCE is a special case and an exception to this rule.) This means that document structures are always created in a top-down fashion. A generic DAP entity can subsequently be linked to another existing generic DAP entity.

Creation functions always return a handle for subsequent reference to the new DAP entity. This may be, for example, for setting properties.

In certain cases, subordinate entities are automatically created together with a new superior DAP entity. For example, when a new generic pageset entity is created, subordinate generic pages are created depending on the PagesetType property. Such a hierarchy of DAP entities, defined by its most superior entity, is called a cluster. The handles of the subordinate entities within the cluster can be used to access properties of those subordinate entities.

In some cases in FOD36, the DAP constraints require that generic entities are placed in one of several possible defined sequences within a Generator for subordinates. This means that an application needs a way of defining this sequence; special functions are provided for this purpose.

In FOD36 some of the additional constituents have SEQ Generator for subordinates constraints, but the application has some choice as to which terms should be included within the Generator for subordinates. Once the choice of terms has been determined, the terms become mandatory. The resulting constraints on the specific structures could not be imposed by the DAP API without complex parameterization of the create functions. However, it is an objective of the DAP API to constrain the application in subsequent specific structure operations such that further instances of the same class (brought about either by dla_duplicate_specific_entity or dla_instantiate_spec_log) conform to the DAP constraints, and in particular that the specific structure matches the Generator for subordinates of the class.

Sub-trees of FOD36 entities that are subject to these constraints are called clumps. There are rules to constrain the production of instances of entities within clumps, so that it is impossible to generate incomplete or inconsistent instances. Further information about clumps is given in 2.2.2.4.

#### 2.1.3.3 Setting DAP Properties

When new DAP entities are created, certain DAP properties and underlying ODA attributes are set automatically, depending on DAP type, API function arguments and so on. Other properties can be set subsequently as a result of explicit function calls to add properties and structure.

The DAP API provides functions to enable individual DAP property values to be set for a specified DAP entity. The application is not permitted to set property values that violate DAP constraints, subject to considerations of performance and usability of the DAP API.

When non-basic values are set, appropriate declarations in the document profile are automatically updated.

In mode-1, when new specific DAP entities are created, additional underlying ODA constituents (classes and styles) can be created to represent them. As DAP property values are set, the appropriate underlying ODA structures are updated. In mode-2 this automatic creation only applies to styles.

When a specific DAP entity is created as a copy of an existing specific DAP entity, all of its DAP properties are automatically applied to the new DAP entity.

#### 2.1.3.4 Writing Content

The DAP API provides functions that enable the application to add content to DAP entities corresponding to basic logical constituents. Content can be text, geometric or raster graphics. Existing content can be added to or overwritten. The DAP API user does not have to be concerned with ODA content portions.

The DAP API does not operate on the content encodings. There is no checking of, for example, character control functions against the set of recorded non-basic values.

### 2.1.4 Reading DAP-conformant Documents

Processable, formatted or formatted processable form documents can be read; these documents can refer to an external document class or to a resource document. The DAP API provides functions to:

- Locate any of the four document roots and the document profile DAP entities.
- Establish a link with an existing external document class or resource document.
- Navigate and locate DAP entities within the specific logical structure and specific layout structure.
- Navigate DAP entities within the generic logical structure and generic layout structure.
- Recognize and identify the use of distinct presentation and layout styles.
- Get the values of DAP properties specified for or applying to DAP entities.
- Read content information from DAP entities.
- Identify which entities in the specific logical and specific layout structures share the same content information.

#### 2.1.4.1 Navigating and Locating DAP Entities

The DAP API provides functions to:

- Find whichever may be present of the specific logical, specific layout, generic logical and generic layout roots.
- Find the document profile.
- Select specific logical and layout DAP entities in forward or reverse sequential order, and to move to superior or subordinate specific DAP entities. In the case of a formatted processable form document, the identity of basic specific logical and layout entities that share content can be determined.
- Locate the generic logical and layout entities and follow the generic structures.

#### 2.1.4.2 Getting DAP Properties

The DAP API provides functions to enable the values of properties applying to individual DAP entities to be read.

It does not provide direct access to ODA attributes. This means, for example, that the application does not have to be concerned with the values of attributes such as Bindings, Content generator or Generator for subordinates expressions, where more convenient DAP properties are defined.

It is possible to read properties as they apply to a specific entity, or to distinguish those properties that are derived from the generic structure and its styles as distinct from the specific structure and its styles.

#### 2.1.4.3 Reading Content

The DAP API provides functions to enable the application to read content from DAP entities corresponding to basic constituents. This can be content accessed via the specific structures, or common content accessed via the generic layout structure.

In the case of formatted processable form documents, it is possible to identify the basic logical and layout entities that both relate to a given part of the content.

The content from any one basic entity can be accessed by a series of reads depending on the buffer size allocated by the application. The DAP API user does not have to be concerned with ODA content portions.

### 2.1.5 DAP API Details

This clause gives detailed information about the following topics:

- DAP entities, DAP types and document structures
- DAP entities and their properties
- Functions for types of properties
- Properties with special functions
- Placement of properties
- Layout directive properties
- Numbering properties
- Reference properties
- Conformance checking
- Monitoring the progress of lengthy operations
- Application callback function

*Note:*
*For each specific function that is identified, a full description can be found in clause 3.*

#### 2.1.5.1 DAP Entities, DAP Types and Document Structures

DAP entities are generally specific or generic, and are referred to by their DAP type; for example a specific DAP entity of DAP type *paragraph*, is called a *specific paragraph.*

See annex B for a full list of the DAP types that are handled by the DAP API. The document profile is included in this list because the DAP API treats it as an entity, with its own DAP type.

There are no DAP types corresponding to content portion DAP constraints. Instances of these constituents can be created automatically as DAP entities are created. They are used when documents containing them are read, but there is no direct access to them.

There are entities corresponding to layout styles and presentation styles, but these can only be manipulated in a limited manner. For instance, styles are created automatically when properties are set on DAP entities that do not already refer to a relevant style, and they cannot be created independently.

The specific logical and layout entities form two hierarchical structures, within which the entities are linked by the ODA attribute Subordinates. It is possible to traverse both of these structures or navigate from entity to entity within them using the navigation functions.

The generic logical and layout entities also form two structures, which in general are directed graphs. It is possible to traverse both of these structures or navigate from entity to entity within them using different properties (not special functions as for the specific structures). Alternatively, the entities within each structure can be visited in random order, or individual entities can be found from their ODIF identifiers, using the functions provided for this purpose.

#### 2.1.5.2 DAP Entities and their Properties

There is generally a one to one correspondence between DAP properties and the equivalent ODA attributes. The properties have the same names as those used in the DAP and in the base standard. Tables showing the codes for these properties and the entities to which they apply can be found in Appendices A and B. Each property has a data type that reflects the structure of the corresponding ODA attribute. The data types are listed in annex A. These types govern the functions used to read and write the properties, as explained in 2.1.5.3.

The ODA attribute Subordinates, which connects specific entities into a hierarchy, is not handled directly as a property. It is maintained automatically as entities are linked together (see 2.2.2.2) and it is interpreted by the navigation functions.

In the case of the generic entities, the ODA attribute Generator for subordinates is maintained for each generic entity as the generic structures are created. During reading, it is interpreted as a set of special properties.

Some ODA attributes are not directly represented at the DAP API, for example, Bindings and Content generators. These attributes underlie DAP properties that represent DAP features more directly.

Any DAP entity, other than the profile, can have the following application supplied annotation properties:

- User readable comments
- User visible name
- Application comments (not available for styles)

The only properties of styles that are directly accessible are the Identification and Annotation properties that correspond to ODA attributes. These properties apply only to the style itself and do not affect any other entity referring to the style. The layout and presentation properties specified, in ODA terms, for the style are only accessible in the DAP API through a logical entity or layout entity that refers to the style. The styles referred to by any DAP entity can be determined and compared with the identity of those referred to by other entities.

Other properties are modelled on the features described in clause 6 of the FODs. The more complex properties are explained in 2.1.5.3 to 2.1.5.8.

#### 2.1.5.3 Functions for Types of Properties

Different functions manipulate DAP properties depending on the type of the property. For a complete list of the functions that are applicable to each type, see annex A. The property types are:

- dla_entity_handle
- dla_entity_type
- dla_structure_handle
- dla_integer
- dla_null
- dla_octet_string
- nonvalued
- SEQUENCE
- CHOICE (dla_integer)

All of these behave as data that can be manipulated, except SEQUENCE and nonvalued. The last three types listed (nonvalued, SEQUENCE, and CHOICE ) are the property types that control two or more subsidiary properties. In general, the manipulation of subsidiary properties is not independent of their superior property.

Properties of the nonvalued type represent a group of subsidiary properties. They can be used to detect the presence of any subset of these subsidiary properties and to remove the complete set by setting the status to null.

A SEQUENCE is a type that modifies the type of a subordinate property. This means that a single property can consist of a repetition of at least one of the other types. This is done by indexing. There are distinct functions that deal with the SEQUENCE or indexed form of the other properties. Tabulation stops collectively form a SEQUENCE property, and have special functions that do not apply to any other property.

The data type dla_structure_handle does not appear on its own but is always in a SEQUENCE. It must therefore always be manipulated by indexing.

The data type dla_structure_handle is used to refer to a collection of underlying more basic types. In most cases, these collections correspond to groupings of parameters and subparameters in ODA attributes.

The behaviour of the CHOICE type is not symmetrical in reading and writing. It represents the use of mutually exclusive underlying attributes in a single property or parameter. When setting properties, the CHOICE type is not used at all and it is only necessary to set the required subsidiary property, using the corresponding property code and function. When reading a property of CHOICE type, it is recommended to first determine which subsidiary property is present, using the function dla_get_choice. This gives a value of dla_integer, which is a code indicating which subsidiary property is present. The corresponding function can then be used to read this property.

The type dla_null is not symmetrical either, because it is associated with the CHOICE type. It is used where one of the mutually exclusive options in a CHOICE type has no value of its own but is merely present or absent. When setting a property of this type, the function dla_set_null_value is used with the property code for the subsidiary property. This is consistent with other CHOICE alternatives except that no value argument is needed with this function. When reading such a property the function dla_get_choice is used as above, but if it gives the option code for a dla_null type, there is no need for any further read operation as the presence of the option has been determined.

An example illustrating the use of dla_get_choice is given in 2.2.13.6, which describes getting new layout objects.

For a complete list of the functions that are applicable to each type, see the introductory clauses of annex A. A simplified version of the function list is given in the remaining paragraphs of this subclause. For examples of their use, see 2.2.8 (and subclauses), which cover setting properties, and 2.2.13 (and subclauses), which cover getting properties.

The following functions are for getting and setting properties of basic types:

| | |
|---|---|
| dla_get_entity_type | |
| dla_get_entity_handle | dla_set_entity_handle |
| dla_get_integer | dla_set_integer |
| dla_get_string | dla_set_string |
| dla_get_spec_entity_handle | |
| dla_get_spec_integer | |
| dla_get_spec_string | |

The following functions are for getting the octet count of properties of string type:

dla_get_length
dla_get_spec_length

The following function is for setting a null-valued property within CHOICE type:

dla_set_null_value

The following functions are for getting an option within properties of CHOICE type:

dla_get_choice
dla_get_spec_choice

The following functions are for getting and setting properties of SEQUENCE (indexed) types:

| | |
|---|---|
| | dla_set_index_null_value |
| dla_get_index_choice | |
| dla_get_index_entity | dla_set_index_entity |
| dla_get_index_integer | dla_set_index_integer |
| dla_get_index_structure | dla_set_index_structure |
| dla_get_index_string | dla_set_index_string |
| dla_get_index_length | |
| dla_get_spec_index_choice | |
| dla_get_spec_index_entity | |
| dla_get_spec_index_integer | |
| dla_get_spec_index_structure | |
| dla_get_spec_index_string | |
| dla_get_spec_index_length | |

The following functions are for getting the cardinality of properties of SEQUENCE (indexed) type:

dla_get_size
dla_get_spec_size

### 2.1.5.4 Properties with Special Functions

The DAP API provides special functions for dealing with tab stops, content information, and character sets. These functions are described here.

There are special functions for inserting and retrieving individual tab stops in association with a basic DAP entity. Each tab is at a specified Basic Measurement Unit (BMU) position with specified alignment and, when relevant, an align-around string.

Tabs exist in the underlying invisible ODA structure, Line Layout Table (LLT). In the DAP API there is a corresponding property of type SEQUENCE that can be used to determine the number of tab stops in an LLT when reading a document. However, two special functions, dla_get_tab_stop and dla_get_spec_tab_stop, can be used to get some basic information about a given tab stop, using its index within the LLT. The special functions dla_get_tab_stop_strings and dla_get_spec_tab_stop_strings can be used to get the remaining information for each tab stop.

When adding tab stops during the writing of a document, no reference is made to the LLT property. Each tab is added to the target basic entity, using the function dla_set_tab_stop. The DAP API Tool adds the tab stop to the invisible LLT that applies to that entity. Tab stops cannot be removed from or modified in an LLT. See 2.2.13.4 for further information about getting tab stops, and 2.2.8.8 for further information about setting tab stops.

There are special functions to manipulate the content data that is associated with basic entities. These functions are described in detail in 2.2.2 (and subclauses), and 2.2.3 (and subclauses).

The content for any one basic entity can be accessed by a series of reads or writes that match the buffer size allocated by the application. The DAP API user does not have to be concerned with ODA content portions.

The content from any basic entity can be converted between any of the alternative encodings for the content architecture of that entity. This can be done within a generic external or resource document, an input or an output document. The content within formatted processable form documents can also be converted. However, the distribution of character content between layout and logical structures is not specially taken into account, so it is necessary to convert all the content within the specific layout structure to ensure a consistent conversion of all character content.

There is no property code for content data, and content is not controlled by rules that apply to other properties. In particular, if the properties of a basic entity are fixed, the content can still be written. If such a basic entity is duplicated, the content is not duplicated. The new duplicated entity has no content until some new content is added to it.

For examples see 2.2.9 (and subclauses), which cover adding content.

There are several properties that indicate the graphic character sets used in different parts of a document. They do this by giving the escape sequences to be used, in octet strings. These properties can be treated directly as octet strings. The DAP API provides several functions to assist with the construction and interpretation of these properties. The function dla_append_char_set can be used to construct, and the functions dla_get_nth_char_set and dla_get_char_set_count (plus dla_get_spec_nth_char_set and dla_get_spec_char_set_count which suppress defaulting) can be used to interpret all properties except the character set property, Non-basic character features.

The function dla_append_char_set aids the construction of the character set escape sequences by generating the appropriate *ISO 2022* character designation sequence from the supplied character set type, code area, and character set id parameters. Appropriate locking shift sequences are always added after the character designation sequence. In the case of the profile property Comments character sets, code extension announcers are added to the property as necessary.

The character set property Non-basic character features is set automatically as the document is built up, provided that the function dla_append_char_set is used. In addition to reading it as an octet string, the functions dla_get_index_char_set and dla_get_index_char_set_count can be used to assist with interpreting

the property. The property Non-basic character features is only set automatically when dla_append_character_set is used. If dla_set_string is used to set the escape sequences directly, the application must also set the non-basic values.

The value of the argument character_set_type specifies whether the character set contains 94 or 96 characters and whether it is single-byte or multi-byte.

The argument code_area specifies one of the sets G0, G1, G2, or G3.

The value of the property Code extension announcers does not need to be set, as all possible valid values are set automatically as the document-wide default.

For examples, see 2.2.13.5, which describes getting graphic character sets, and 2.2.8.9, which describes setting graphic character sets.

#### 2.1.5.5 Placement of Properties

In the ODA standard the attributes that apply to an object can be specified for that object itself, or can be specified for a related style or class. The relationship between the object to which the attribute applies and the constituent for which it is specified may be even more distant in cases where resource documents and default value lists are used.

There are various ways in which an attribute can apply to an object, or in DAP API terms in which a property can apply to an entity. For example, it can be specified directly for an entity, on a related class or style, in a default value list or in the profile defaults. The DAPs impose some constraints to this flexible placement of properties, so the full range of possibilities is not available for all properties and entities. However, examples of most of the variations are available within FOD36.

The DAP API allows the application to exploit most of these different forms of property placement, and accepts conformant documents. It also adds two different forms of placement of properties that do not correspond directly to ODA variants. These two forms are discussed in this subclause together with the standard forms.

The handling of property placement by the DAP API has two objectives: to enable an application to ignore most of the permitted variations, or to enable it to exploit them, and to assist with the correct and conformant usage of the applying values.

In the writing direction an application can choose one of several modes in which to write the document. When writing the logical structure, using mode-1 makes it possible to set all properties as if they were specified for the specific logical entities, whilst the DAP API assumes responsibility for specifying the corresponding attribute either for that entity or for a class or style constituent. In any writing mode, an application writer can make a more complete use of the permitted variations, by assigning styles explicitly to entities other than the one for which they were created. There are also functions that produce some sharing of classes and styles, and thus of the attributes that are specified for them, between an indefinite number of entities.

In the reading direction the distribution of attributes between constituents has already been established by the originating system; an application can make use of this information or ignore it at the discretion of the application writer. When use is made of the defaulting mechanism provided by the DAP API (partly supported by the ODA API), it is also possible to determine the constituent from which the information was derived. As explained in detail in 2.2.3.3, which covers defaulting DAP properties, it is also possible to restrict the reading of properties to those specified for the entity that is being read and for a style that is directly referenced by that entity.

In some cases, when both reading and writing, properties do not appear on the entity corresponding to the constituent for which the attribute is specified. Examples of this are Footnote numbering and (generic) Page dimensions. In both of these cases it is necessary either for correct operation or for conformance that some attributes must match on two or more constituents. To facilitate this, a superior entity is chosen to be the interface to the properties corresponding to the matching attributes. In the case of Page dimensions, each page has an orientation property to show that although it must have the same dimensions as the other pages in the page set, it may (in FOD36) have a different orientation.

There are two special cases involving the defaulting process as provided by the DAP API that are additional to those already explained. These are defaulting as provided for generic entities (generic layout entities in particular) and defaulting of content information. These cases are explained in 2.2.3.3 which covers defaulting of properties.

#### 2.1.5.6 Layout Directive Properties

The naming of layout directive properties is based on clause 6 of the FODs rather than on the names of ODA attributes. For this reason, the properties are explained in ODA terms in this subclause. There are also references to the parts of clause 6 that address the document features.

In ODA and in the FODs several layout directive attributes offer the choice of indicating the layout object involved by its class identifier or, for lowest level frames, by the use of a layout category. Layout categories provide additional flexibility because several frames can have the same (permitted) category, and each frame can have several (permitted) categories. The DAP API simplifies this complexity by assigning and using layout categories automatically, and by hiding and interpreting them at the interface. The use of categories for column layout requires different permitted categories for the different columns. These category names are also generated automatically. However, the corresponding Layout in column property must be used on all basic logical entities in order to obtain the correct layout. The null layout category cannot be used.

As well as generating the Permitted categories attribute on lowest level frames automatically, the DAP API uses the correct layout category when a basic logical entity is given one of the Layout in column, Footnote layout, Layout in single FormEntryArea, Layout in Cell or Layout in basic float properties. Similarly, when reading an input document, the Layout category attribute is interpreted by looking up the lowest level frame that has the corresponding Permitted categories attribute. If there is more than one such frame, then any one of these can be supplied.

The layout of content into footnote areas is also controlled by the use of layout categories. For FOD26 there can only be one category of footnote area, so this aspect is controlled automatically by the DAP API. For FOD36 there may be multiple categories of footnote area, so this aspect cannot be controlled automatically by the DAP API, and must be controlled by the application in the same way as for layout in columns. The approach is to assume that there is a single footnote area and all new footnotes in a FOD36 document use the FOD26 footnote category (unless an external document class with generic layout directives is being used). If multiple footnote areas are to be used, then the layout directive Footnote layout (DLA_C_LYD_FNOTE_FRM) must be specified separately. This layout directive must also be used for any ReferencedContent entities in a footnote, whether or not multiple footnote areas are being used.

Where several frames have or are to have the same category, this can be detected by the use of distinct properties or established by the use of the function dla_make_frame_equiv respectively. The condition where any frame has more than one permitted category cannot be established when writing documents nor detected when reading documents.

It is possible to modify the automatic layout category allocation. If content is to be laid out continuously in recto verso paginated documents, there could be a single frame (or set of frames in the case of snaking column layout) that is linked to the body frame of both pages. In this case no special action need be taken and a single layout category will suffice. Alternatively, if there are to be different frames on the recto and verso pages that receive the same text, the permitted categories automatically allocated to one frame, or set of frames, must be altered to be the same as that allocated to the other. The function dla_make_frame_equiv alters the permitted category of one frame entity to be the same as another.

For examples relating to the information given in this subclause, see 2.2.13.6, which describes getting new layout objects, 2.2.8.4, which describes setting page breaks, and 2.2.8.5, which describes setting column breaks.

#### 2.1.5.7 Numbering Properties

The numbering properties relate solely to page numbering at level 1 (FOD11), additionally to section and footnote numbering at level 2 (FOD26) and are quite general at level 3 (FOD36). Except for in the FOD36 case, all the numbering properties are named after their context, so that at levels 1 and 2 there are no general numbering properties that apply to more than one of the three types of numbering. Segment numbering, and the more general numbering for FOD36, is more complex because of its hierarchical nature and it is

explained separately within this subclause. Because of the relationship between the FODs, FOD36 retains the simple schemes of lower levels as well as adding its more general alternatives.

The numbering properties are complex properties that are represented in ODA by two different attributes: Bindings and Content generators. These attributes are not independent, but must correspond in their Binding identifiers and the way in which they are used. The DAP API Tool generates appropriate values of these ODA attributes when creating documents, and interprets them when reading documents. In this way, it is not necessary for an application to be concerned with the identifiers of the Bindings used.

The DAP API does not provide special functions to set or get numbering properties. The functions that are used are also used for other properties and are described in 2.1.5.3. For examples of the use of these functions for numbering, see 2.2.8.2, which describes setting values for segment numbers, and 2.2.13.3, which describes getting values for segment numbers.

There are two varieties of numbering properties: numbering value properties and numbering format properties. The former can be specified at or above the hierarchical level of the entity at which they are to be effective. The latter can only be specified for the numbered entity at which they are to be effective.

Numbering formats include the option to use the counting mechanism known in ODA as ORDINAL or ORD, (see *IS8613-2* or *CCITT T.412* clause 5.1.3.2). This option should be used with extreme care. It is appropriate to only a few contexts, such as simple segment numbering or non recto-verso page numbering. When used with recto-verso pages it causes the recto pages to be numbered separately from the verso ones, so that pairs of them have the same numbers. It also cancels the effects both of numbering initialization and of resetting.

There is a further distinction made, for numbering properties other than those for page numbering, between initialization and resetting. The initialization properties are placed on the classes of the entities for which they are specified. This means that they are shared and apply to all instances of those classes. The resetting properties are placed on the specific entity for which they are specified and are not shared. The resetting properties are available for the Document logical root, but as there can be only one specific and one generic Document logical root, the distinction is not significant.

In FOD36, page and footnote numbering can be like segment numbering in FOD26, in that there is an arbitrary number of separate sequences of counts, each one identified by an integer tag (the <n> in number - <n>). These integer tags are in the same series as the segment number tags, because the same binding name stem is used. Unlike the FOD26 segment numbering, there is no link between the tagged counters and the hierarchical level of the footnote in the logical structure, or of the page in the layout structure. For compatibility with FOD26, the plain, un-indexed page and footnote counters are also available, but incrementation of the footnote counter must be specified explicitly via a special-purpose property.

For documents created to conform with FOD11 and FOD26, the automatic incrementation within a sequence of numbers is provided by underlying binding expressions which are generated automatically by the DAP Level API. Such bindings are also generated automatically if Segment Numbering properties are specified in FOD36 documents. For General number properties in FOD36 note that the incrementation bindings are not established automatically and must be specified explicitly via a special-purpose property.

It is recommended that the FOD26 properties are used in FOD36 documents for simplicity. The mechanism incorporated in the FOD26 property scheme for mapping segment levels to the corresponding binding names works for FOD36 documents provided that any use of explicit binding name tags is arranged to avoid the automatically generated ones. If this cannot be ensured, or assumed in the case of externally produced documents, then the full FOD36 numbering scheme must be used, and the FOD26 scheme restricted to FOD26 documents. The DAP Level API cannot ensure that a mixed use of mechanisms (ie FOD36 and FO26 numbering properties on the same entity) will work correctly.

Note that the FOD constraints rule that all number binding values are incremented before use. Initial and reset values of numbers are therefore one less than the lowest number imaged.

Page and footnote numbering properties are detailed in annex A. The list of these properties is as follows:

Initial page number
Page number format
Page number string prefix
Page number string suffix
Page number string function

Initial footnote number
Footnote prefix
Footnote suffix
Footnote format
Footnote string
Footnote string Function

Reset footnote number

Segment numbering value properties are held as an indexed sequence of structures on numbered segments and their superiors. Each structure in the sequence refers to one level of numbered segment.

The segment numbering properties are detailed in annex A. The list of these properties is as follows:

| | |
|---|---|
| Initial segment level number | Reset segment level number |
| Initial segment value | Reset segment value |
| Initial segment prefix string | Reset segment prefix string |
| Initial segment suffix string | Reset segment suffix string |
| Initial segment separator string | Reset segment separator string |

The level number is derived from the level of numbered segment to which the numbering applies, with the first level encountered below the document logical root being level one.

The initialization values are those that are given the same name in the FODs.

Reset segment value and the reset strings are for altering the automatic numbering scheme within a sequence of sibling segments.

The DAP API provides properties to govern segment number format. The relevant varieties of property are:

Segment number string
Segment number expression type
Segment number string function
Segment number format
Segment number strings required
Segment number special prefix
Segment number special suffix

The DAP API also provides properties (for FOD36 only) to govern general number format. The relevant varieties of property are:

Numbering identifier
General number string
General number string function
General number format
Inherit string identity (for hierarchic numbering format)
General number strings required
General number special prefix
General number special suffix

The general numbering properties (for FOD36 only) are:

| | |
|---|---|
| Initial numbering identifier | Reset numbering identifier |
| Initial value | Reset value |
| Initial prefix string | Reset prefix string |
| Initial suffix string | Reset suffix string |
| Increment value | |

#### 2.1.5.8 Reference Properties

In FOD36 there are several DAP entity types provided specifically to introduce cross-references from one part or parts of a document to another. They only allow for character content references, but these can be

included either as part of the specific logical structure of the document, or as common content to be laid out into particular frames deriving from the generic layout structure.

The basic entities among these DAP types can have complex properties for setting up the values and formats of the generated content. This process can make use of both the numeric values of other sections, footnotes etc. (with formatting choice independent of their origin) and, in the case of hierarchical section numbering, of the possibly concatenated text form of these numbers.

As well as references to other properties based on ODA bindings, there is a special string binding for use in reference entities. This string is intended to be initialized to an arbitrary sequence of octets in the same way as prefixes and suffixes, and can then be invoked in various places. So chapter headings, for example, can be reiterated on all page headers containing parts of that chapter.

The cross-references can be between nominated entities in the specific logical structure, or indirectly to layout objects such as pages and hence page numbers that contain the logical entity.

#### 2.1.5.9 Conformance Checking

Conformance checking is invoked for every call to the DAP API during document creation. Additional checking is done when the properties of an entity are fixed and when the document is written.

The DAP API does not attempt to ensure that a document is formattable without errors.

The DAP API does not check any aspects of content information for conformance.

When reading a document, the DAP API generally requires that the document conforms to a supported DAP as indicated in the document's profile. An error return due to non-conformance may be returned when opening a document as the DAP API checks that the details of the document profile describe a document that is supported by the DAP API.

During the execution of dla_read_document or of dla_read_generic_doc, an error return due to non-conformance may be returned in the following cases:

- Values of document profile attributes indicate that the document is not within the restrictions imposed by the DAP API.
- An attribute value is such as to prevent correct interpretation of a number of other attributes into conformant properties.

The DAP API prevents the specification of attribute values for a constituent that are not permitted by the DAPs by disallowing the setting of corresponding property values in DAP entities.

The DAP API ensures that:

- All mandatory constituents are present.
- All mandatory attributes of each constituent are present.
- The value of each attribute is in the range of permissible values for that attribute.

The DAP API does not check, for instance, that:

- The document can be formatted.
- The content information conforms to DAP constraints.
- The attributes Character path, Line progression and Character orientation of logical objects are consistent with the page layout type of the page into which the logical object is meant to be laid out.
- A footnote frame exists on the page on which the footnote reference is to be laid out.

#### 2.1.5.10 Monitoring Progress of Lengthy Operations

On certain platforms, the functions dla_write_document and dla_read_document may take an appreciable length of time to execute, particularly when processing large documents. For this reason, the DAP API offers the facility for an application to register a callback function. Only one callback function can be specified for each instance of the DAP API Toolkit.

The callback function is supplied by the application, and is invoked periodically by the DAP API during the reading and writing operations. The callback function must match the function prototype given in 2.1.5.11, and is invoked with two parameters. The first parameter is the handle of the Toolkit instance invoking the callback function; the second parameter is an integer in the range 0-100, and is an estimate of the degree of progress through the read or write operation, expressed as a percentage.

It is intended that the application-supplied callback function is used to display a progress indicator for the possibly lengthy reading and writing operations. The precise nature of this progress indicator is environment-dependent. When used interactively, a callback function can provide the opportunity for a user to terminate the read or write operation prematurely. The callback function also provides an opportunity for the application to yield the CPU in non-preemptive scheduling systems (for example, an application written for Microsoft Windows might include a standard message loop within the callback function).

The application-supplied callback function is registered with the DAP API by means of the function dla_register_callback_function.

To prevent the callback function being called out of context, the function can be unregistered by supplying a NULL function pointer to dla_register_callback_function. It is recommended that this is done when the read or write operation is complete.

If the application does not require progress reports or the opportunity to yield the CPU, then it is not necessary to supply or to register a callback function.

*Note:*
*Calling DAP API functions from within the application-supplied callback function is neither supported nor recommended. As this function is only called within the context of dla_write_document or dla_read_document, invoking other DAP API functions within the callback function may have unpredictable and undesirable effects.*

### 2.1.5.11 Application Callback Function

This subclause describes the form of the application-supplied callback function that can be called periodically during dla_write_document and dla_read_document. The precise functionality of the callback function is under the control of the application programmer.

When a Windows application wishes to use the DAP API callback facility, the following must be carried out:

- The application's callback function must be declared with the "far pascal" keywords.
- The application's callback function must be exported in the "EXPORTS" section of the application's module definition file.
- The function pointer passed to dla_register_callback_function must be a Windows procedure-instance address, and not the address of the callback function itself. This procedure-instance address is formed by calling the Windows API function MakeProcInstance, passing it the address of the application's callback function.

**C Synopsis**

```
dla_integer             application_callback_function(
    dla_toolkit_handle      toolkit_handle,
    dla_integer             percent_complete);
```

**Description**

This function is invoked periodically by dla_write_document and dla_read_document, and is intended to give the application the opportunity to display some dynamic indication of the progress of these operations, which may take an appreciable length of time to execute. When used interactively, a callback function can provide the opportunity for a user to terminate the read or write operation prematurely. The opportunity is also presented for the application to yield the CPU on non-preemptive operating systems. The name of this function is not important as it is invoked via a function pointer.

The parameter toolkit_handle gives the handle of the DAP API Toolkit instance that is invoking this callback function.

The parameter percent_complete gives the estimated progress through the current operation, expressed as a percentage from 0 to 100.

The application must always return from this function if the read or write operation is to complete. The application must return an integer value, that is one of:

| Value | Meaning |
|---|---|
| DLA_C_CONTINUE | Continue processing the read or write operation |
| DLA_C_STOP | Stop processing the read or write operation immediately and return the error DLA_E_CALLBACK_ERROR to the application |

*Note:*
*Calling DAP API functions from within the application-supplied callback function is neither supported nor recommended. As this function is only called within the context of dla_write_document or dla_read_document, invoking other DAP API functions within the callback function may have unpredictable and undesirable effects.*

**Arguments**

**toolkit_handle**
The handle of the DAP API Toolkit instance that is invoking this callback function.

**percent_complete**
An integer value in the range 0 to 100 indicating an estimate of the progress of the current read or write operation, expressed as a percentage.

**Results**

**Status**
Normally DLA_C_CONTINUE to indicate that the read or write operation is to continue; otherwise, if the application determines that execution of the read or write operation is no longer required, the callback function can return the value DLA_C_STOP to indicate that the operation should be terminated immediately.

## 2.2 Writing Applications

This subclause provides an introduction to writing applications. It also provides examples of the use of the DAP Level API in both the reading and writing direction. It does not show the complete processing of a document, and there is no coherent processing structure inherent in the examples.

Status checking code has generally been omitted from the examples, in order to make them easier to read. This omission is not recommended in practice. The exception to this is for reading and writing documents where an illustration of the examination of failures is included.

Initialization and termination are introduced in 2.2.1.

Introductory information about creating documents are given in 2.2.2 (and subclauses). The topics covered are:

- Creating and linking generic DAP entities
- Creating and linking specific DAP entities
- Creating and linking passages and numbered segments
- Creating and linking clumps

- Setting DAP properties
- Fixing DAP properties
- Unfixing DAP properties
- Duplicating DAP entities
- Writing content

Introductory information about reading documents are given in 2.2.3. The topics covered are:

- Navigating and locating DAP entities
- Getting DAP properties
- Defaulting DAP properties
- Reading content

The writing direction is covered in 2.2.4 to 2.2.9 (and subclauses). They illustrate the use of the DAP Level API when creating an ODIF data stream that conforms to a DAP level. Examples are provided to illustrate the use of the DAP Level API in the following areas:

- Toolkit and document management operations
- Creating generic and specific logical entities
- Creating generic layout entities
- Property setting
- Adding content

The reading direction is covered in 2.2.10 to 2.2.14 (and subclauses). They illustrate the use of the DAP Level API when reading a DAP-conformant document that may reference a generic document. Examples are provided to illustrate the use of the DAP Level API in the following areas:

- Toolkit and document management operations
- Navigating specific structures
- Navigating generic structures
- Property getting
- Reading content

### 2.2.1 Initialization and Termination

An instance of the DAP Level API Tool must be initialized before performing any document processing, and terminated after completing document processing.

The DAP API provides functions to create a new DAP document or to commence the reading of an existing DAP document. The DAP level of the document is established at this time. The level cannot be changed, and it must be within the set of DAPs supported by the version of the Tool in use. As well as specifying the required DAP level when creating a new DAP document using the dla_create_document function, the mode of operation must also be selected. Information about modes of operation is given in 2.1.3.1.

The DAP API provides a function to write a copy of a DAP document as an ODIF data stream. Further document processing can continue after this copy has been written. ODIF data stream constructs, such as ODA identifiers, are automatically completed.

The DAP API provides a function to delete a DAP document in order to relinquish the resources that it occupies. If the deleted document refers to a generic document, then the generic document is deleted as well.

When a pre-existing document or generic document is read, it is placed in a read-only state. Changes cannot be made to a document while it is in this state. The read-only state of a pre-existing document does not prevent it from being deleted when it is of no further use. A generic document can only be deleted by deleting its specific document.

For examples, see 2.2.4, which gives information on Toolkit and document management operations, and 2.2.10 to 2.2.14, which give examples of the reading direction.

*Note:*
*Other restrictions that are inherent in the DAP API Tool may cause a document to be rejected when it is read. See 2.3 for information about restrictions and limitations.*

### 2.2.2 Creating Documents

The dla_create_document function creates a new DAP document independent of any other document that is accessible to the current Toolkit instance. It requires the user to specify a DAP level and to select the mode of operation, in order to establish, for example, whether the generic logical structure is to be created explicitly. The mode is established when the document is created and cannot subsequently be changed, with one exception which is covered later.

Certain DAP entities are created automatically when a DAP document is created, for example, entities representing a document profile and, depending on the selected mode, structure roots. Structure roots are only created automatically in mode-1, because in other modes the document may either use, or be intended to be used as, an external generic document.

The application makes calls explicitly to create DAP entities comprising the complete generic layout structure, unless an external document is used. There is no call to create the generic layout root. In mode-1, this happens when the document is created, whereas in mode-2 and mode DLA_C_FOR_EXTERNAL the generic layout root is created when the first generic PageSet is created.

In mode-1 the application makes calls explicitly to create DAP entities comprising the specific logical structure and a corresponding generic logical structure is created automatically . If mode-2 has been selected, the application makes calls explicitly to create DAP entities comprising the generic logical structure, and to establish every subordinate relationship. The application then explicitly creates each DAP entity within the specific logical structure as an instance of its generic entity. The function dla_change_writing_mode can be used (providing no FOD36 clumps are partly formed) to alter the mode from mode-1 to mode-2, but no other mode change can be made.

The construction of each of the three structures is normally performed in an interleaved fashion. Generally, the only constraint limiting this interleaved method of construction derives from the fact that an entity must exist before it can be referenced, and in mode-2 each specific logical entity must refer to its class at the time of its instantiation. In FOD36 documents some logical entities form clumps, which gives rise to additional constraints. Additionally, a layout directive property can only reference a part of the generic layout structure that has been created.

As an alternative to explicit creation of the generic logical structure and generic layout structure, both can be acquired by reading a generic document as an external document class. This means that the creation of documents that refer to a pre-existing external class is supported.

The creation of a document structure consists of the following:

- Creating DAP entities and linking them into the structures
- Setting DAP properties on newly created DAP entities
- Writing content to DAP entities
- The creation of the specific layout structure is not supported.

According to the FOD constraints, resource documents can only contain basic components. The DAP API functions require that all structures are created in a top-down manner. The creation of resource documents is an exception so as to be conformant with DAP rules. To facilitate the creation of resource documents, the Resources attribute in the document profile has a corresponding DAP API property, and the automatic assignment of object identifiers to generic entities is inhibited so that the application must define them.

For examples, see 2.2.4 (and subclauses), which describe creating documents, and 2.2.5 to 2.2.7 (and subclauses), which describe creating entities.

#### 2.2.2.1 Creating and Linking Generic DAP Entities

The DAP API provides functions to create a new generic DAP entity of a given DAP type. In mode-1 this applies only to layout DAP types and to those for common content, because the generic logical types are created automatically. In mode-2 it applies to both logical and layout DAP types.

In modes DLA_C_FOR_EXTERNAL and DLA_C_FOR_RESOURCE the application must supply the ODIF identifiers for all entities, including styles.

Creation functions always return a handle for subsequent reference to the new DAP entity, in order, for example, to set properties.

When a new generic DAP entity is created it is always (except in mode DLA_C_FOR_RESOURCE) linked as a subordinate to an existing generic DAP entity, such as the appropriate document root. This means that document structures are always created in a top-down fashion.

A generic DAP entity can subsequently be linked to another existing parent DAP entity. The Parent entities property of generic entities is therefore a SEQUENCE property, because there may be several parents or superiors. Multiple linking of this sort permits the reuse of a single generic entity, possibly with a set of subordinate generic entities, in several contexts, such as on recto and verso pages. This results in more compact ODA documents.

The linking of generic DAP entities implies that the DAP API automatically amends the Generator for subordinates expression in the underlying ODA classes. Linking is not permitted if it violates DAP constraints.

In mode-1, the linking of generic logical DAP entities takes place automatically as the application establishes the specific logical structure. In other modes, the linking of generic logical DAP entities must be done explicitly by the application, or it is not permitted at all.

In some cases in FOD36, the DAP constraints require that generic entities are placed in one of several possible defined sequences within a Generator for subordinates, and an application therefore requires a way of defining this sequence. Special functions are provided for this purpose.

In certain cases, subordinate entities are automatically created together with a new superior DAP entity. For example, when a new pageset is created, subordinate pages are created depending on the property, PagesetType. Such a hierarchy of DAP entities is defined by its most superior entity and is called a cluster. The handles of the subordinate entities within the cluster exist as properties of the superior. They can be used to access properties of those subordinate entities. The subordinate entities cannot be created or linked separately. The following generic layout clusters exist:

- Generic pageset, with subordinate pages
- Generic multicolumn frames, with subordinate column frames

***Note:***
*In FOD36 documents, additional composite frames can be either created or linked as subordinates to a snaking or synchronized columns frame, within the possibly empty sequence of subordinate frames that has been created as part of the cluster.*

The following generic logical clusters exist:

- NumberedSegment
- Footnote
- Reference

The following functions create generic layout entities:

```
dla_create_generic_body_frame
dla_create_generic_frame
dla_create_generic_frame_seq
dla_create_generic_fnote_area
dla_create_generic_basic_float
dla_create_generic_head_footer
```

dla_create_generic_pageset
dla_create_generic_snaking
dla_create_generic_synchronized

The following functions create generic logical entities:

dla_create_gen_log_entity
dla_create_gen_log_entity_seq

For examples of these functions see 2.2.5, which covers the creation of generic logical entities and 2.2.7 (and subclauses), which cover the creation of generic layout entities.

The following function creates a generic common logical entity:

dla_create_generic_comm_entity

The following functions link two generic entities:

dla_link_generic_entity
dla_link_generic_entity_seq

*Note:*
*In the case of FOD36 documents there is a constraint that governs the creation of some generic logical entities relative to the creation of instances of those entities. Further information on this topic is given in 2.2.2.4, which covers creating and linking clumps.*

### 2.2.2.2 Creating and Linking Specific DAP Entities

The DAP API provides functions to create a new specific DAP entity of a given DAP type, or as a copy of an existing specific DAP entity. The functions that can be used depend on the mode that has been selected. In the modes for creating resource and external generic documents, a specific logical structure cannot be created at all.

In mode-1 the two functions are:

dla_create_specific_entity
dla_duplicate_specific_entity

In modes other than mode-1 the process of creating specific entities is called instantiation to emphasize the fact that an instance of a generic DAP entity is being created. It is not permitted to have a specific entity that is not an instance of an existing generic entity, or to change this entity to another one.

In modes other than mode-1 the two functions are:

dla_instantiate_spec_log
dla_duplicate_specific_entity

When a new DAP entity (other than a root entity) is created, it is always linked as a subordinate to an existing DAP entity, such as the appropriate document root. This means that document structures are always created in a top-down fashion. This parent child relationship cannot subsequently be changed.

Linking for specific DAP entities can be specified in the following ways:

- In relation to a parent entity, preceding or succeeding all siblings
- Preceding or succeeding a specified sibling

Creation functions always return a handle for subsequent reference to the new DAP entity, in order, for example, to set properties.

In certain cases subordinate entities are automatically created together with a new superior DAP entity, thus creating a cluster. For example, when a new footnote is created, subordinate entities are also created, including a footnote reference entity. Such a hierarchy of DAP entities is defined by its most superior entity and is called a cluster. The handles of the subordinate entities within the cluster exist as properties of the superior. They can be used to access properties of those subordinate entities. The subordinate entities cannot be created or linked separately.

The following specific logical clusters exist:

- NumberedSegment, with subordinate Number
- Footnote, with subordinate FootnoteReference, FootnoteBody and FootnoteNumber
- Reference, with subordinate ReferencedContent

The difference between using dla_create_specific_entity and dla_duplicate_specific_entity is explained in 2.2.2.8, which covers duplicating DAP entities. It is especially significant in mode-1 and in the case of Passage and NumberedSegment entities. (See 2.2.2.3 for information about Passages and Numbered Segments). This is because using dla_duplicate_specific_entity causes entities to share their classes, and in the case of these two types of entity the DAP constraints restrict subordinate classes. The result is that the subordinate subtrees of duplicated entities of these two types have to have the same numbering format for their NumberedSegment entities. Where this similarity is desired, as is often the case, using duplication in mode-1 makes it unnecessary to re-specify the numbering formats within the subtrees.

For examples, see 2.2.6 (and subclauses), which cover the creation of specific logical entities.

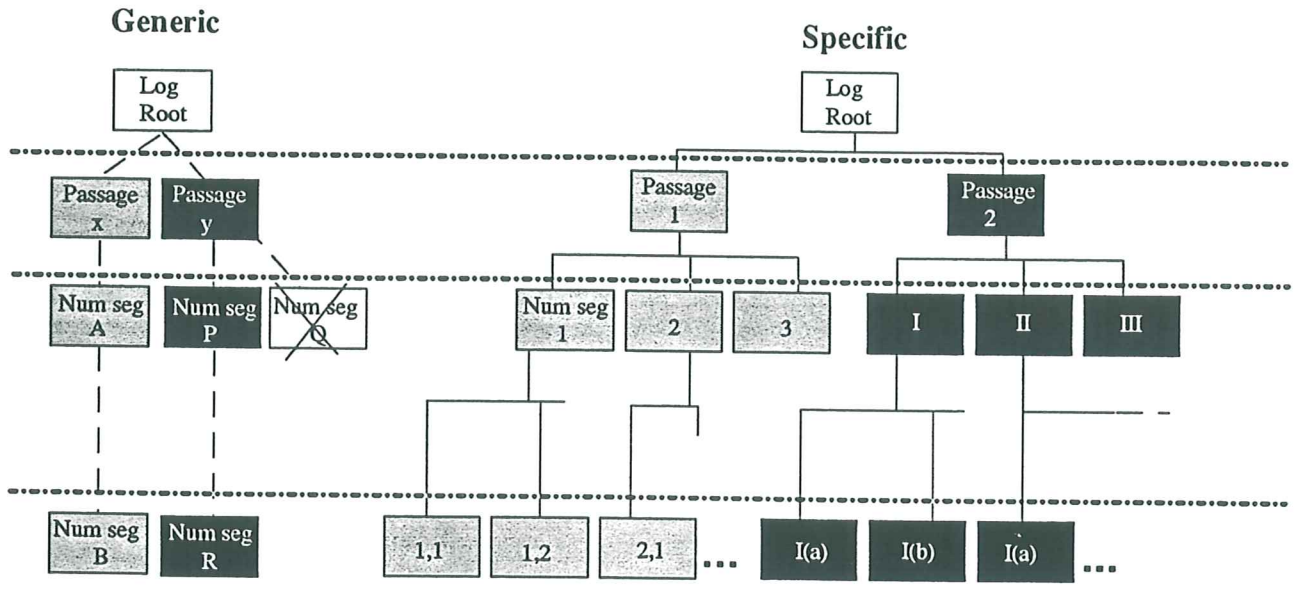
*Note:*
*In the case of FOD36 documents there is a constraint that governs the creation of some generic logical entities relative to the creation of instances of those entities. Further information on this topic is given in 2.2.2.4, which covers creating and linking clumps.*

### 2.2.2.3 Creating and Linking Passages and Numbered Segments

There are particular constraints on the ways in which Passages and Numbered Segments can be created and linked. These constraints derive from the constraints imposed by FOD26 and FOD36.

For documents conforming to FOD26 the constraint is that there can only be one class of NumberedSegment corresponding to each successive depth of numbering within any one Passage. Figure 1 illustrates this constraint; the DAP API does not permit NumberedSegment class Q to be linked subordinate to Passage Y, as class P already occupies this position.

*Note:*
*In figures 1 to 5, horizontal dash-dot lines indicate the different depths applying to entities in the generic and specific structures, and the way in which the boxes are filled indicates the association of generic with specific entities. So in figure 1, for example, specific NumberedSegments I, II and III are at depth 2 and are instances of the generic NumberedSegment P.*




*Note:*
*Other constituents, even mandatory subordinates such as Number, are omitted from the diagrams for simplicity.*

**Figure 1 - Constraints on Linking Generic NumberedSegments**

Once a generic NumberedSegment has been created and its depth established (by virtue of its linked position when created) it cannot be linked elsewhere in the document, using dla_link_generic_entity or dla_link_generic_entity_seq, except where it would have the same depth. Figure 2 illustrates this constraint; NumberedSegment class B occurs for the third depth of numbered Segment under both classes A and P. NumberedSegment class A cannot be re-used subordinate to B, as its depth is already established as depth 2 by its earlier occurrence in the structure. The new position would require it to have a depth of 4.

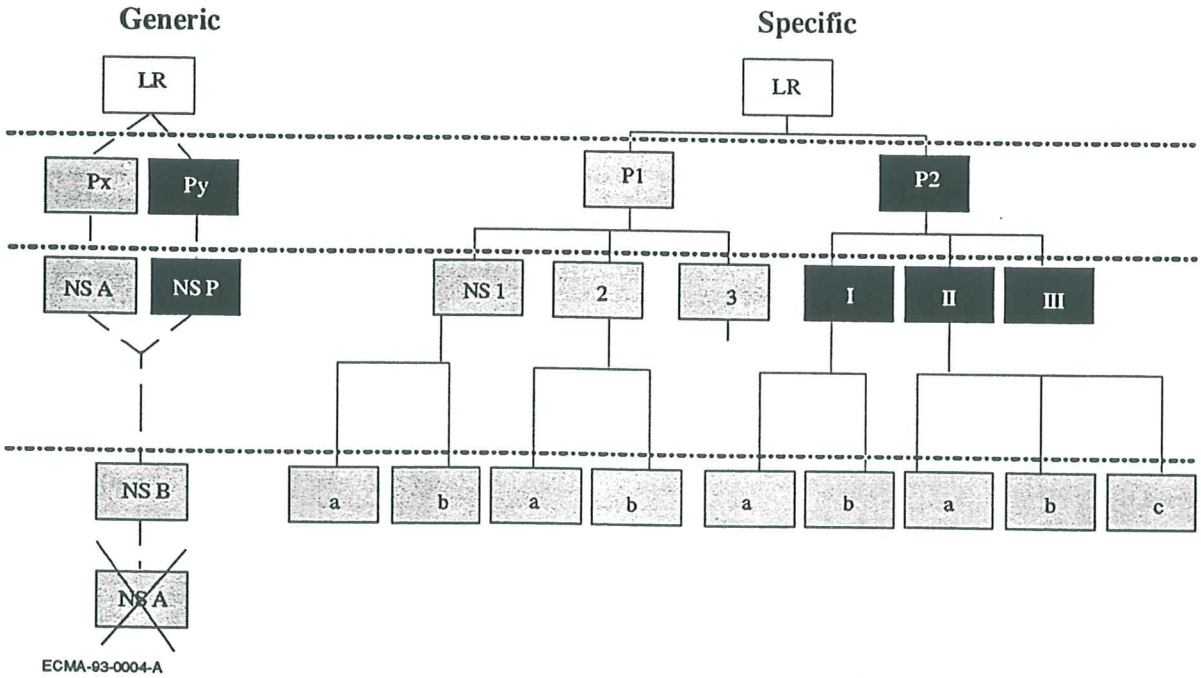



**Figure 2 - Constraints on Linking Generic NumberedSegments**

The information relating to figures 1 and 2 is in terms of mode-2, in that it discusses creating and linking generic entities. In mode-1 similar restrictions apply but are apparent when creating or duplicating specific entities. The underlying constraint, that Generator for subordinates attributes of Passage and NumberedSegment classes can only contain a reference to a single NumberedSegment class, is the same, but a class is created for each new specific NumberedSegment, except where this is not allowed. In either mode, a NumberedSegment cannot be duplicated into a position where the depth of the new NumberedSegment entity would be different from that of the model entity.

The reason for these constraints concerns the Bindings and Content generator attributes of each generic NumberedSegment and Number. These contain binding names that control the automatic numbering scheme for the given depth and must be employed consistently .

The value of the depth is 1 for the NumberedSegment immediately below a Passage in documents conforming to FOD26 and increases for each level below that.

For documents conforming to FOD36, the situation is more complex because of the greater choice provided by that DAP. A NumberedSegment can occur immediately below the Logical Root, and Passages can be placed subordinate to a NumberedSegment, with further subtrees of NumberedSegments below them. The generic Logical Root and any generic NumberedSegment or Passage, at whatever depth within the document, can have at the most one generic NumberedSegment subordinate to it. The definition of the value of the depth is that it is 1 for any NumberedSegment that is the first to be found in descending from the root, and it increases for each level below that. Where a Passage occurs within a logical subtree, which is subordinate (even indirectly) to a NumberedSegment, the counting of depth continues as if the Passage were not present.

Given this definition of depth, similar constraints to those applying to FOD26 documents also apply to FOD36 documents. Once the depth of a generic NumberedSegment has been established by the position at which it is initially created, (directly in mode-2 or indirectly in mode-1), the DAP API prevents the linking of this generic NumberedSegment, or the use of instances of it, at a different depth anywhere within the document.

A single generic NumberedSegment can be created subordinate to any Passage, but if a single existing generic NumberedSegment is to be linked in such a position, then it must be at the same depth.

Such placement may come about either directly through use of dla_link_generic_entity or dla_link_generic_entity_seq in mode-2, or indirectly as a result of a dla_duplicate_specific_entity call in mode-1.

Figure 3 shows how, in a FOD-36 document, a Passage can be placed in the structure beneath a NumberedSegment. The depth of the NumberedSegment labelled 2.3 is depth 3. Passages of class T take the same depth as the superior NumberedSegment, and the numbering of any subordinate NumberedSegments continues on from the previous ones at that depth. Figure 3 also shows how, by linking the same class of NumberedSegment (NS Z in Fig 3) beneath both the Passage class T and the NumberedSegment class Y, the format of numbering beneath Passages of class T continues unchanged from the previous NumberedSegments at that level.

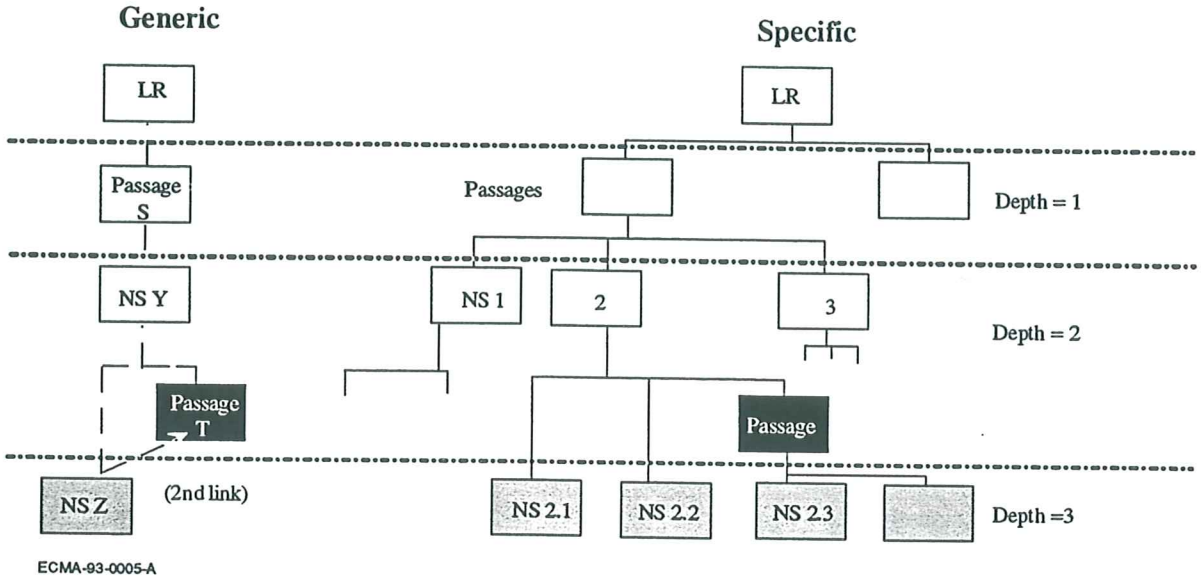



**Figure 3 - Placing Passages (FOD36 only)**

A generic Passage is marked as having a certain depth as soon as a generic NumberedSegment is placed directly or indirectly subordinate to it. The depth is one less than that of the subordinate NumberedSegment (thus Passages that have no superior NumberedSegment are of depth 0). Generic Passages can be linked elsewhere in the document only if their depth is undefined, because they have as yet no subordinate NumberedSegment classes, or if their depth is consistent with the new position in the structure. Such placement may come about either directly through use of dla_link_generic_entity or dla_link_generic_entity_seq in mode-2, or indirectly as a result of a dla_duplicate_specific_entity call in mode-1.

It follows from this that a generic Passage that has been linked so that it occurs at two or more different depths (a second link having been permitted implying the absence of subordinate NumberedSegments) cannot subsequently have any generic NumberedSegment placed subordinate to it, however indirectly, because its depth cannot be defined unambiguously.

This means that automatic segment numbering, using the FOD26 scheme within a FOD36 document, may increment continuously through several specific NumberedSegments, although they may be subordinate to instances of different generic Passages and thus of different classes. In FOD36 only, this permits such NumberedSegments to take different formats whilst retaining the numbering.

In figure 4 the central block of black NumberedSegments uses a different numbering format, because the black NumberedSegments are instances of different classes to those of the surrounding grey NumberedSegments. However, the numbering has been carried through the black NumberedSegments.

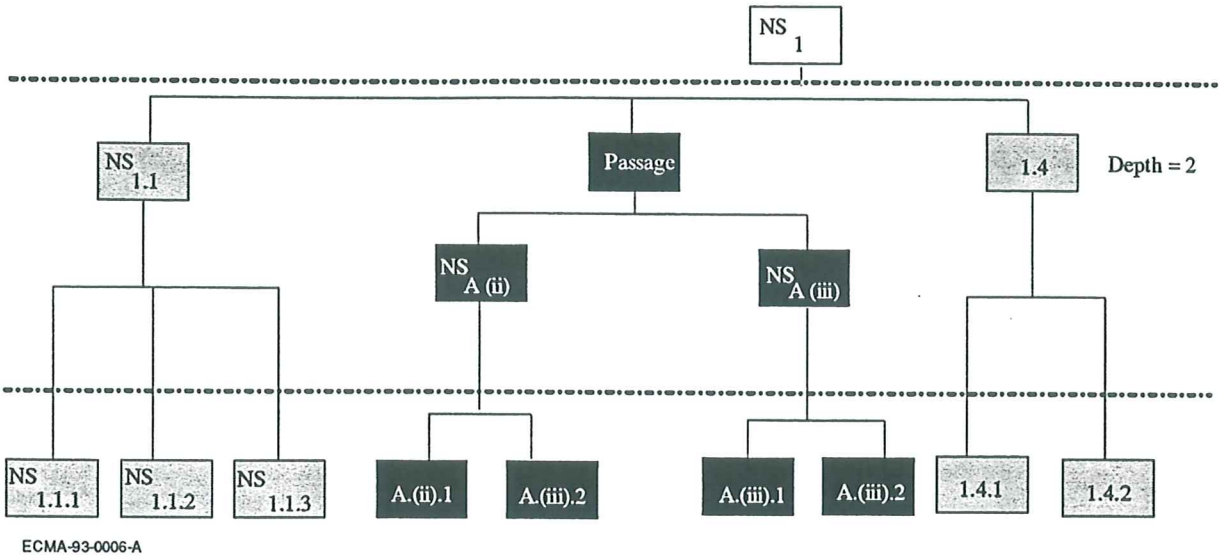



**Figure 4 - Continuous Incrementation of Automatic Segment Numbering (FOD36 only)**

It also follows that sub-structures of Passages incorporating DAP types that do not include NumberedSegments can be used without any structural constraints other than the fundamental ones that derive from the FOD36 specifications for Generator for subordinates.

If a truly independent numbering scheme is required inside a nested Passage, such that the automatic numbering of NumberedSegments surrounding the new subtree is not affected, it is necessary to use the general numbering properties that are only available for documents conforming to FOD36, as this is the only way to arrange binding name variants for these entities that are independent of the depth.

In figure 5, the central block of black NumberedSegments uses a numbering scheme that is independent of the scheme used by the neighbouring grey NumberedSegments, with different formats and independently incremented numbers.

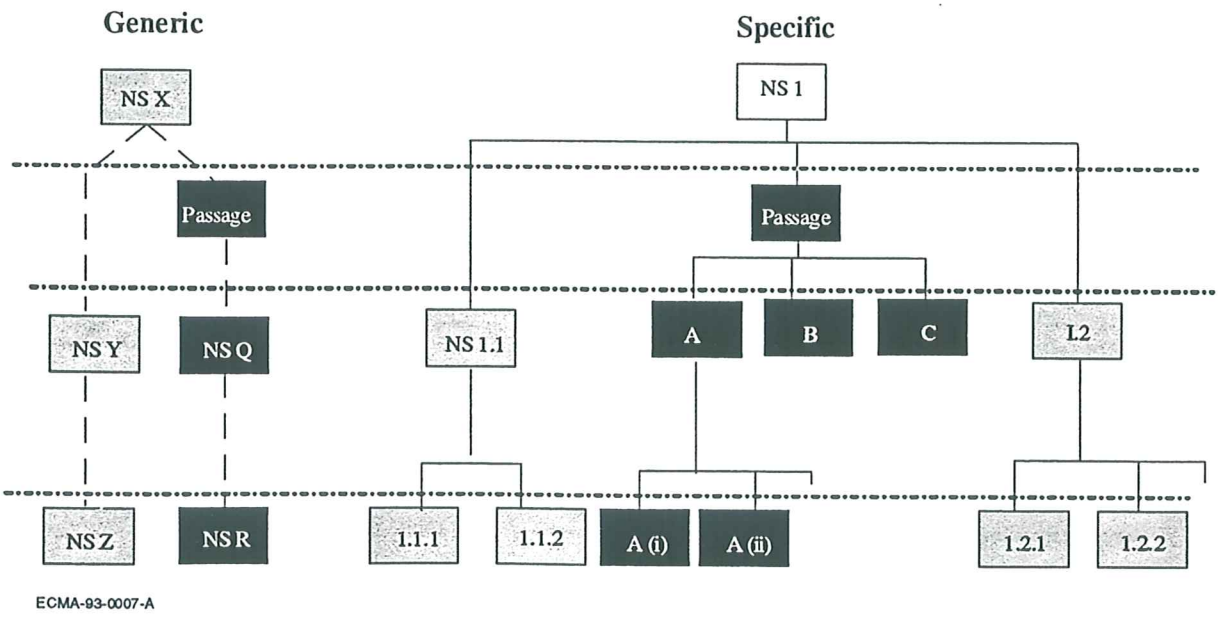



**Figure 5 - Interrupted Incrementation of Automatic Segment Numbering (FOD36 only)**

### 2.2.2.4 Creating and Linking Clumps

In FOD36 documents there can be sets of related logical entities called *clumps*. The logical entities that make up a clump are less closely related than those within a cluster, but more closely related than other

entities. A clump is a set of entities for which the Generator for subordinates in the generic logical structure may include certain terms. It is not mandatory to include these terms, but once a term has been included in the Generator for subordinates, it becomes mandatory for that clump and makes the presence of the corresponding subordinate specific entities mandatory.

The presence or otherwise of these terms in the Generator for subordinates for a given generic clump is determined in one of the following ways:

- In mode-1, by such dla_create_specific_entity calls that are made up to the time of the first dla_duplicate_specific_entity call. That is, while there is only one instance of the entity and its subordinates that make up the clump.
- In mode-2, by such dla_create_gen_log_entity(_seq) calls or such dla_link_generic_entity(_seq) calls that are made up to the time of the first dla_instantiate_spec_log call. That is, while there are no instances of the clump.

Once the structure of the clump has been determined, the clump behaves as a cluster, but only in so far as the creation of instances is concerned. This is because the terms that have been included in the Generator for subordinates, and the entities corresponding to those terms, become mandatory once this determining process is complete. The DAP API prevents the placement of further subordinates that would require the Generator for subordinates to be updated after the structure of the clump has been determined.

It is an objective of the DAP API to constrain the application in subsequent specific structure operations such that further instances of the same class, brought about either by the dla_create_specific_entity, dla_duplicate_specific_entity or dla_instantiate_spec_log functions, conform to the DAP constraints. Once the structure of a clump has been determined, the dla_duplicate_specific_entity or dla_instantiate_spec_log functions create the whole clump; that is, the complete set of mandatory subordinate entities together with the superior entity of the clump.

For the entity types NumberedList, DefinitionList and, in certain cases, UnNumberedList, the subordinate entities must be created in pairs. The DAP API does not, in this case, create any subordinates together with the superior entity, but when subordinate entities are created, they can only be created in pairs. This can be done by requesting either one of the entities making up the pair.

There are rules that constrain the production of instances of entities within clumps such that it is impossible to generate further instances that are inconsistent or incomplete.

In the following text, which describe the different types of clumps, the information refers to the facilities and constraints in the two principal modes of operation of the DAP API, that is, mode-1 and mode-2. In both these modes the application is undertaking the creation of a complete document, so it is relevant to consider the creation of both specific and generic logical structures. In other modes, the document being created is not a complete document, so discussion of some operations is not relevant. However, the behaviour of the DAP API for operations such as dla_duplicate_specific_entity and dla_instantiate_spec_log in mode DLA_C_WITH_EXTERNAL generally reflect the protocols described for mode-2.

In each mode some of the function calls involved are different, but the situations are equivalent. There is a first phase, in which the shape of the sub-tree that comprises the clump is determined by the application, followed by a second phase, where the shape of the clump is fixed and subsequent function calls result in a cluster of entities coming into being.

The following text describe the characteristics and constraints on methods of creating the various entity types that are clumps.

**EntryElement**
EntryElement is only allowed one subordinate, which is of one of the DAP types BodyText, BodyRaster or BodyGeometric. Once the application causes the subordinate class to be linked to a generic EntryElement and the structure of the clump is determined, that class of EntryElement behaves as a cluster in that the DAP API creates the subordinate with each new instance.

**Form and EntryGroup**

The possible subordinates of Form or EntryGroup are an arbitrary set of entities of types EntryGroup and EntryElement. A Form can exist only as a subordinate of Figure, and an EntryGroup can exist only as a (possibly indirect) subordinate of a Form, so all of these entities must be part of one or more Figure clumps.

Until the structure of the generic clump (or in mode-1, the first specific clump and its corresponding generic) is determined, all operations cause a single entity to come into being.

The function dla_create_specific_entity can only be used for entities of type Figure, Form, EntryGroup, EntryElement or any of their subordinates, to build up the first instance. In this context, dla_create_specific_entity always causes the creation of a new class for any of the entity types.

Once the structure of the Form or EntryGroup is determined, any further instances of the same class must have the same complete set of coordinates. In mode-2, such further instances can only be created by calling dla_duplicate_specific_entity or dla_instantiate_spec_log, citing the parent specific or generic Figure entity respectively, and not any of its subordinates. In mode-1 only, it is possible to duplicate a subordinate of a clump whose structure has been determined, if the new entity will be subordinate to a new Figure, Form or EntryGroup whose structure has not yet been determined.

The possible subordinates of Form or EntryGroup consist of a set of entities in any order (ODA Generator for subordinate construction type AGG). The constitution of this set is determined as explained above. As the term AGG means that instances must consist of exactly one of each of the constituents whose classes are in the AGG expression, the DAP API prevents the creation of documents where this is not the case. Although the order of the instances is nominally free, the DAP API insists on the instances all being in the same order as each other, and the same as the order in which the corresponding classes appear in the Generator for subordinates. One class may appear several times in a Generator for subordinates as subordinate to another class, in which case there must be the same number of instances. For further information on this subject see 2.3 which covers restrictions.

The DAP API does not allow direct or indirect recursion of EntryGroups, so an EntryGroup class cannot appear in its own Generator for subordinates, or in that of any of its directly or indirectly subordinate classes. For the same reason, the DAP API does not allow an EntryGroup to be duplicated subordinate to itself, or to any of its directly or indirectly subordinate EntryGroups, or to any other instances of the classes of such EntryGroups.

**Figure**

Entities of type Figure have one mandatory subordinate that can be either an Artwork or a Form. This must be present before a specific entity of type Figure can be duplicated, or a generic entity of type Figure can be referenced for a dla_instantiate_spec_log function. If a Figure has a subordinate Form, the structure of the Figure is not determined until the structure of the subordinate Form is determined.

When the structure of the Figure has been determined it includes the Form, so that when the Figure is duplicated or an instance of the Figure is created, the Form is duplicated or created as well. If a Figure has a subordinate Artwork which is not itself a clump, the Figure clump includes only itself and its immediate subordinates.

Figure can have up to three other subordinates, one each of type Number, Caption and Description, which, if determined to be present, are created with the Figure on subsequent dla_instantiate_spec_log or dla_duplicate_specific_entity function calls.

Once the type of the mandatory subordinate and the presence of the other discretionary subordinates is determined, no additional entities can be placed subordinate to the Figure.

**Table**

Because the DAP API employs the most permissive variety of Generator for subordinates expression that is allowed by the DAPs for each case, entities of type Table consist of a succession of Rows, of the same or different class.

Each Row is constructed from one of the following:

- An arrangement of EntryElements
- A fixed sequence of a single EntryElement and a TableComponent

If the application requests a TableComponent to be linked subordinate to a Row, there must be no more than one EntryElement already linked. As soon as a TableComponent is placed, the Generator for subordinates that will eventually be generated will be a SEQ. The Row is now a clump, to the extent that it has mandatory subordinates.

Each EntryElement is also a clump, in that when it is instantiated or duplicated, the single subordinate is created by the DAP API. (This behaviour also applies to EntryElements subordinate to entities of type Form.)

Entities of type Table are not in themselves clumps, but individual components of Tables may be clumps. In particular EntryElements behave as clumps, and if a Row contains a TableComponent the Row too is a clump. The clump entities within a Row are only its immediate subordinates; the process does not involve indirect subordinates in this case.

**NumberedList**
The subordinate entities of NumberedList must be created in pairs, the pair consisting of a Number and a ListItem. There can be any number of subordinate pairs, but the first pair is not created when a NumberedList is created.

Either one of a pair can be cited as the class in a call of dla_instantiate_spec_log or as the model_entity in a call of dla_duplicate_specific_entity, but the pair is always created together. If another pair is linked before or after an existing pair, it must be positioned either before the first entity or after the second entity of the existing pair. That is to say, it must not be positioned so as to split up the two entities of the existing pair.

Second or subsequent pairs of Number and ListItem placed under a NumberedList must be of the same class as the first. This can be arranged by calling dla_duplicate_specific_entity citing the first pair as model_entity, or calling dla_instantiate_spec_log citing the same class. If dla_create_specific_entity is used, no further classes are created.

Additional classes cannot be linked under a generic NumberedList.

**DefinitionList**
DefinitionList is the same as NumberedList, except that ListTerm replaces Number.

**UnNumberedList**
UnNumberedList is similar to NumberedList in that when it is created, the subordinates are not created at the same time. There can be up to two subordinates: a separator, which is of one of the DAP types BodyText, BodyRaster or BodyGeometric, and ListItem. ListItem is mandatory and the separator is optional.

Once the structure is determined, subsequent dla_instantiate_spec_log or dla_duplicate_specific_entity operations on the UnNumberedList cause the same set to come into being; that is, either the single ListItem or the pair of a separator and ListItem. Further subordinates can be added as follows. If the separator subordinate is present, the pairs of subordinates (separator, ListItem) are constrained to be created and placed in pairs. If another pair is linked before or after an existing pair, it must be positioned either before the first entity or after the second entity of the existing pair. That is to say, it must not be positioned so as to split up the two entities of the existing pair. The separator and ListItem must have the same class as the initial pair. This can be arranged by calling dla_duplicate_specific_entity citing the first pair as model_entity, or calling dla_instantiate_spec_log citing the same class. If dla_create_specific_entity is used, no further classes are created.

Additional classes cannot be linked under a generic UnNumberedList.

#### 2.2.2.5 Setting DAP Properties

The DAP API provides functions for setting individual DAP property values on a given DAP entity. The application must not set property values that violate DAP constraints. When non-basic values are set, the appropriate properties to declare this are automatically updated in the document profile . For further information about the setting functions, see 2.1.5.3, which describes functions for types of properties, and 2.1.5.4, which describes properties with special functions.

When using mode-1, the only entities that can be the subject of functions to set properties are the profile, specific logical entities, generic layout entities and, in a limited manner, styles. In mode-1, when DAP properties equivalent to ODA attributes of logical classes are specified, the DAP API automatically specifies

the attributes for the class of the entity. In mode-2 such properties must be specified for the generic entity directly. The same applies (for FOD36 documents where the only examples occur) where attributes are constrained by the DAPs to be specified for styles that are specified only for classes and not for specific objects.

When DAP properties equivalent to ODA attributes in presentation styles or layout styles are specified, and no style is already invoked, the DAP API automatically manages the creation of and reference to appropriate styles. The automatic management of styles makes it possible for applications to be written without considering the mapping of properties to attributes on presentation styles versus layout styles. For more information about automatic style management, see 2.2.2.6, which covers fixing DAP properties, 2.2.2.7, which covers unfixing DAP properties, and 2.2.2.8, which covers duplicating DAP entities.

Alternatively, application writers may choose to distinguish whether properties correspond to attributes in layout styles or in presentation styles. In this case, once a presentation style or layout style has been created automatically as described above, it can then be applied individually to other new entities that do not already have a style of that type. This process is constrained by the DAP provisions for the application of styles to DAP types. In FOD36 documents, appropriate automatically created styles can also be specified for the properties that correspond to the ODA attribute Default value lists (DLA_C_DVL_PRES_STYLE and DLA_C_DVL_LAY_STYLE).

ODA provides several defaulting mechanisms that invoke information specified for constituents closely related to a specific object (such as a style or class) and also for those related more remotely (such as in a resource document or document profile). Because of the complexity of the defaulting rules and the way in which they influence the attributes applying to many ODA constituents simultaneously, the DAP API makes no attempt to verify the conformance of properties that are specified remotely rather than through a directly related class or style, and in particular, in FOD36 documents, in Default value lists.

When a style is created and attributes are specified for it by assigning properties to a basic entity, conformance checks are applied for each property. If a FOD36 document is being created, this style may then be specified for a Default value lists property of a composite entity, but no further conformance check is applied as FOD36 does not indicate any constraint.

When new DAP entities are created, certain DAP properties and underlying ODA attributes are automatically set depending on DAP type, API parameters and so on. An application cannot set or alter such properties. Others are set subsequently as properties and structure are added.

For example, the following underlying ODA attributes are set, in various circumstances:

- Application comments (to reflect constraint-name)
- Object type
- Object class
- Generator for subordinates
- Subordinates (of superior)
- Permitted categories (on FootnoteArea and column frames)

When a specific DAP entity is created as a copy of an existing DAP entity using dla_duplicate_specific_entity, all DAP properties of the existing DAP entity are applied to the new entity.

The DAP API does not provide functions to remove properties, for instance to remove tab stops from an LLT.

For examples involving property setting, see 2.2.8.

#### 2.2.2.6 Fixing DAP Properties

The ODA standard allows for various mechanisms whereby a single specification of an attribute can apply to several constituents. This sharing of data is reflected in the DAP API, as is explained in 2.1.5.5, which gives information on the placement of properties. Because of the complexity of the mechanisms, and the rules and constraints that apply, the DAP API provides for several alternative degrees of control that application writers can use to assist them in arranging for the sharing of data between entities.

As a consequence of the sharing of data between entities, any subsequent changes to a shared property of any of these entities would be applied to the other entities that share the property. This would become a problem where the resulting change was not intended. Such changes are prevented by setting entities that are to remain unaltered into a fixed state by using the function dla_fix_properties.

To avoid any possibility of unwanted accidental modification, the properties of each entity should be fixed, using the function dla_fix_properties, as soon as all of the entity's properties are established. This prevents accidental modification of the properties, if the properties of another entity that share data from the first entity are modified. The DAP API Tool achieves this prevention either by making copies of the data to remove the sharing or, where such copying is precluded by DAP constraints, by rejecting the modification.

While an entity is in the fixed state it is not possible to change any of its properties. Additional conformance checks for self-consistency and completeness are therefore applied when a fix is done. It would be both inefficient and cumbersome either to perform these checks every time a property was set, or to defer the checking until the document processing was complete. This means that when the properties of a DAP entity are declared fixed, an error response resulting from the additional conformance checks may be generated. It is possible to unfix an entity, change any of its properties and then fix it again. This process is explained further in 2.2.2.7, which covers unfixing properties. Each time an entity is fixed there is a check for basic conformance of its properties.

There is no property code for content data and content is not controlled by fixing. If a basic entity is fixed, the content associated with it can still be written. The same applies to linking within the generic and specific structures; subordinate entities can be created and linked whether or not the entities involved are fixed.

Extensive use of the function dla_duplicate_specific_entity results in the maximising of data sharing in the document representation, within the DAP constraint rules. Because duplication is intended to introduce sharing of data, applications must fix the properties of any entity before duplication is permitted. This is referred to as the *fix protocol*.

The fix protocol is mandatory in all modes that permit the creation of specific logical entities. Before a newly created specific logical entity can be duplicated, its properties must be fixed using the function dla_fix_properties. Any entity that exists as the result of a duplication is automatically in the fixed state and can also be duplicated.

In mode-1 the generic logical entities are effectively read-only, in that their properties can only be altered by operating on a specific instance of each entity. When a specific logical entity is fixed, the properties that correspond to attributes of the class are effectively fixed, as well as those that belong to the styles, which are also read-only.

In mode-2 the generic logical entities are not fixed when their instances are fixed, and they are only fixed when this is requested explicitly. Any accidental modification of properties due to the explicit sharing of styles is prevented by fixing generic logical entities, and some checking for conformance can be invoked before calling dla_write_document. When developing a mode-2 application, there is a possible benefit to be gained for both conformance checking and the trapping of accidental updates, by fixing generic logical entities. However, the fix protocol does not apply in the case of generic logical entities because there is no duplicate function that can be applied to them. Generic documents that are being used as resource documents or external document classes cannot have their generic logical entities fixed because they are both read-only documents.

Generally, when a new specific DAP entity is created (not using the function dla_duplicate_specific_entity) none of the properties are fixed. However, in mode-1 this is not always the case and some properties of such an entity may be fixed. DAP constraints, especially the FOD26 and FOD36 constraints on NumberedSegment classes, enforce the sharing of properties such as segment numbering properties. If no NumberedSegments have their properties fixed then accidental updating of these shared properties could occur and would not be trapped. It is more secure to protect against this by fixing properties. The DAP API then rejects any attempt to alter one of the shared properties of another entity. There are distinct error codes to indicate when this has happened. This situation does not arise when using modes other than mode-1, because generic logical entities are either explicitly controlled by the application or are read-only. In either case, their properties are always distinct from those of specific logical entities.

For examples, see 2.2.6 (and subclauses), which cover creating specific logical entities.

#### 2.2.2.7 Unfixing DAP Properties

While an entity is in the fixed state it is not possible to change any of its properties. However, it is possible to unfix an entity, change any of its properties and then fix it again. Each time an entity is fixed there is a check for basic conformance of its properties.

As the purpose of the fixing mechanism is to avoid any possibility of accidental modification of the properties of an entity when the underlying attributes are shared, the DAP API endeavours to maintain this protection when an entity is unfixed. This provision differs for mode-1 and mode-2 because of the different responsibilities that the Tool and application have in these two modes.

In mode-1, when a specific logical entity is fixed, the properties that correspond to attributes of the class are effectively fixed, as well as those that are specified for the styles. If the specific entity is then unfixed, the class and styles are only unfixed if they (taken individually) are not shared with any other fixed entity. If there is no sharing, as may be the case if the entity was created and not duplicated, then the unfix reverses the effect of the fix and properties can be altered freely. If there is sharing but the other entities are all unfixed or not yet fixed, then properties can still be altered freely and will apply to all the sharing entities. These are the only circumstances in which properties that belong to the class can be altered.

In mode-2 the generic logical entities appear at the API and are fixed and unfixed quite separately from their specific logical entities. When a specific logical entity is fixed, only the properties that belong as attributes of itself and its styles are effectively fixed. This also applies to fixing a generic logical entity. If either sort of entity is unfixed, the styles are only unfixed if they (taken individually) are not shared with any other fixed entity. If there is no sharing then the unfix exactly reverses the effect of the fix and properties can be altered freely. If there is sharing but the other entities are all unfixed or not yet fixed, then properties can still be altered freely and will apply to all the sharing entities.

In any mode, if the property that is to be altered after an entity has been unfixed is specified as an attribute of a style that is shared with another entity that is still fixed, then a new style is created by copying the original one and the alteration takes place using this new style. An unfixed entity can therefore be altered, but this may result in an increase in the population of styles and a reduction in the instances in which they are shared.

In any mode, applications have to fix the properties of an entity before duplication is permitted. However, duplication is not the only way in which sharing of styles can be introduced, because styles can be assigned explicitly. The assignment of styles can only be done to an unfixed entity because the Presentation style and Layout style properties are controlled by the fixed state. The assigned style can be fixed (because it is in use by a fixed entity) or not fixed (because it is shared only by unfixed entities). Thus the protection mechanism of the fixed state is imposed in such circumstances in a consistent way.

For an example, see 2.2.6 4, which illustrates the use of dla_unfix_properties.

#### 2.2.2.8 Duplicating DAP Entities

The function dla_duplicate_specific_entity creates a new specific logical DAP entity and causes all the properties of an earlier entity to apply to the new one.

Before an entity can be duplicated it must be fixed, as explained in 2.2.2.6. When an entity is created by duplicating an existing fixed entity, the new entity is in the fixed state immediately.

If a basic entity is duplicated, the content is not duplicated. The new duplicated entity has no content until some new content is added to it. Although the properties of a duplicated basic entity are fixed, the content can still be written without having to unfix the entity first.

In mode-1, the difference between using dla_create_specific_entity and dla_duplicate_specific_entity is especially significant for Passage and NumberedSegment entities. This is because using dla_create_specific_entity causes a new class to be created (in most cases), whereas using dla_duplicate_specific_entity causes entities to share their classes. In the case of these two types of entity in particular, the FOD26 and FOD36 constraints place restrictions on subordinate classes. The result is that the subordinate subtrees of duplicated entities of these two types must have the same numbering format for their NumberedSegment entities. Where this similarity is desired, as is often the case, using duplication makes it unnecessary to re-specify the numbering formats within the subtrees.

In mode-2, the same restriction on duplicating NumberedSegment entities applies because the original and new entities have the same class. However, because the hierarchy of numbered segment classes must be explicitly defined by the application, the use of the duplication function does not influence the specification of numbering formats.

*Note:*
*Duplicating entities that are clusters results in the creation of multiple new entities. Duplicating entities that are clumps leads to further complexities that are explained in 2.2.2.4, which covers creating and linking clumps.*

For examples, see 2.2.5 and 2.2.6, which cover the creation of logical entities, and 2.2.13.3, which covers getting values for segment numbers.

#### 2.2.2.9 Writing Content

The DAP API provides functions for the application to add content to DAP entities corresponding to basic constituents. These DAP entities can either be contained in the specific logical structure, or attached to the generic layout structure as part of the common content.

Content can be text, geometric graphics, or raster graphics. Only one content portion is created for each basic logical object.

When writing content, an offset must be specified. This is the number of octets from the first octet belonging to the basic entity. Other parameters of the functions specify the length of the buffer and data in octets. This means that content can be manipulated using buffers of any size. The application can work out the required offset for successive write operations, or can read the current length of the content in octets. The function dla_get_content_length can be used to read the length of the content, but as defaulting is not appropriate, the following functions are recommended:

dla_add_content
dla_get_spec_content_length

There is no property code for content data, and content is not controlled by rules that apply to other properties. In particular, if the properties of a basic entity are fixed, the content can still be written. If such a basic entity is duplicated, the content is not duplicated. The new duplicated entity has no content until some new content is added to it.

For examples, see 2.2.9 (and subclauses), which cover adding content.

*Note:*
*The DAP API does not operate on the content encodings. There is no checking of character control functions, for instance, against the set of recorded non-basic values. This is an application responsibility.*

### 2.2.3 Reading DAP-conformant Documents

When reading or writing documents of appropriate architecture, the DAP API supports the following:

- Locating the document profile and from one to four root entities
- Establishing a link with a pre-existing external document class or resource document
- Navigating the generic logical structure
- Navigating the specific logical structure
- Navigating the generic layout structure
- Navigating the specific layout structure
- Getting properties of DAP entities

For examples, see 2.2.10 to 2.2.14, which cover the reading direction.

Any reading operation can be performed when writing a document.

#### 2.2.3.1 Navigating and Locating DAP Entities

The DAP API provides functions to locate the generic and specific logical and layout roots, and the document profile. These functions are:

dla_move_to_root_entity
dla_find_profile

The DAP API provides navigation functions to scan through the full set of generic logical and layout entities in arbitrary order. These functions are:

dla_find_first_entity
dla_find_next_entity

The DAP API provides a function to find a generic logical or layout entity using its unique ODIF identifier. This function is:

dla_locate_gen_entity

The DAP API provides navigation functions to move to specific logical and layout entities in forward or reverse sequential order, and to move to superior or subordinate specific entities. These functions are:

dla_move_to_child_spec_entity
dla_move_to_parent_spec_entity
dla_move_to_sibling_spec_entity

There are property acquisition functions to select subordinate generic logical and layout entities according to their distinct relationships, and to select superior generic entities. These functions are:

dla_get_entity_handle
dla_get_index_entity

In the case of formatted processable form documents, it is possible to identify the basic logical and layout entities that both refer to a given part of the content. This function is:

dla_get_other_parent

For examples, see 2.2.11 (and subclauses), which cover the navigation of logical structures, and 2.2.12 (and subclauses), which cover the navigation of generic structures.

#### 2.2.3.2 Getting DAP Properties

In general, attributes and styles in the ODA architecture can be found on a specific object (entity) or a generic object class (generic entity), or on styles that are referenced from either of these sorts of entity.

The presentation styles and layout styles in a document are visible as entities, but can only be manipulated in a limited manner. The only properties of styles that can be read directly are the identification and annotation properties corresponding to ODA attributes. These properties apply only to the style itself and do not affect any other entity referring to the style. The layout and presentation properties that are specified, in ODA terms, for the style are only accessible in the DAP API through a logical entity or layout entity referring to the style.

There are two distinct ways in which most properties can be read: with defaulting and without defaulting. The mechanisms for and implications of defaulting are described in 2.2.3.3.

The DAP API provides the function dla_get_status so that an application can verify whether a property applies to or is actually specified for an entity. It indicates the handle of a generic logical entity where the property is either specified on that entity or on a style referred to by it. Similarly, other property reading functions have parameters for determining where properties are specified.

The DAP API does not provide direct access to Bindings or Content generator expressions; an alternative simplified version is provided. There are also special purpose properties that represent defined parts of some Generator for subordinates expressions.

See 2.2.13 (and subclauses) for examples of the property getting functions.

#### 2.2.3.3 Defaulting DAP Properties

In general, attributes and styles in the ODA architecture can be found on a specific object (entity) or a generic object class (generic entity), or on styles that are referenced from either of these sorts of entity.

When reading a document, an application can choose to read the value of properties that apply to entities. In this case, but for specific entities only, the ODA defaulting process is carried out and the location of the specification of the ODA attribute is generally returned as well as the value. A similar process is available for cases where there is no specific entity, in which the ODA defaulting rules found in *ISO 8613-2* clause 5.1.2.4 from point c) onwards are applied. Alternatively, the defaulting process can be inhibited by using a different reading function call, in which case a value is only returned if the property is specified for the target entity or a style associated with it.

Corresponding with this, there are two distinct ways in which most properties can be read: with defaulting and without defaulting. Functions with the syllable _spec_ in their names are used for reading properties without defaulting. For properties read with defaulting, the applying values, wherever they are located, are used to determine the property value. For properties read without defaulting, a value is only supplied if there is a specification for the corresponding attribute on the constituent underlying the entity, or on a style to which it refers. See 2.1.5.3 and 2.1.5.4 for lists of property reading functions.

The functions that inhibit defaulting include _spec_ in their function names. If a function either with or without _spec_ in its name is used with non-defaultable properties, they both return the same value or status. Functions that do not have an alternative that includes _spec_ in the name, such as dla_get_entity_type or dla_get_index_char_set, either do not perform any defaulting at all or the DAP constraints ensure that the returned value does not depend on the defaulting process.

The use of defaulting when reading properties from generic entities gives rise to one case where the returned value for the location of the specified property value may not be immediately obvious. This is when reading the list of common content entities from generic frames such as BasicHeaders and several others. In this case, the handle of the CommonContent entity is returned, as the list of common content entities corresponds to the Generator for subordinates attribute of this entity.

The defaulting process that is performed is not always exactly the same as that specified in the ODA standard. There are several reasons for the differences and each reason gives rise to a distinct form of difference. In the case of properties that are identified in annex A as being based on a single ODA attribute, the behaviour is generally exactly as indicated in the ODA standard. The exceptions are given in the following list of properties whose derivation to the default value as given in the standard is too complex or context-dependent to be derived by the ODA or DAP level APIs:

- Type of coding — DLA_C_CP_TY_COD
- Identifier start offset (of First line format), — DLA_C_CA_ITM_STR_OFF
- Initial point horizontal (ODA attribute: Initial offset) — DLA_C_CA_INIT_PNT_HRZ
- Initial point vertical (ODA attribute: Initial offset) — DLA_C_CA_INIT_PNT_VRT
- CGM text rendition individual part — DLA_C_GA_TXT_RND_IND
- Clipping second X coordinate — DLA_C_RA_CLP_SND_X
- Clipping second Y coordinate — DLA_C_RA_CLP_SND_Y
- Page offset (ODA attribute: Page position) — DLA_C_PGE_OFF
- Fixed dimension — DLA_C_FRM_FIX_DMS
- Horizontal dimension — DLA_C_FIX_DMS_HRZ
- Vertical dimension — DLA_C_FIX_DMS_VRT
- Path dimension Fixed dimension — DLA_C_PTH_DMS_FIX
- Orthogonal dimension Fixed dimension — DLA_C_OGL_DMS_FIX

In the case of the properties listed above, the DAP API may return a status of unspecified even though the properties are otherwise defaultable, if there is no value specified in the document. It is then a user responsibility to derive a suitable default value. The argument derived_location returns a value of DLA_C_DEF_STD_DEF (see 2.4) for these cases, together with a value of DAL_C_PROP_UNSPECIFIED for the argument property_status.

In cases such as the DAP API character presentation property First line format, which involves the interpretation of three separately defaultable ODA attributes, the defaulting of the property can only reflect that of the underlying attributes if all three are specified for the same constituent. The value of the arguments indicating the source of the data reflects at least one of the underlying attributes, but which one is not determinable. If the property is read without defaulting then none of the three attributes are read with defaulting by the DAP API.

The properties for initialization and resetting of binding values are also special cases. They correspond to the same bindings attribute, but initialization applies to generic entities whereas resetting applies to specific entities, so the defaulting behaviour is as if the underlying properties were separate.

In the case of reading content from basic entities, a special defaulting mechanism is provided. This reflects the behaviour of the corresponding ODA attribute for the layout and imaging processes. The same process does not apply to the use of the function dla_convert_content.

#### 2.2.3.4 Reading Content

The DAP API provides functions for the application to read content from DAP entities corresponding to basic constituents. These DAP entities can either be contained in the specific logical structure, or attached to the generic layout structure as part of the common content.

Content can be text, geometric graphics, or raster graphics. Boundaries between content portions are not apparent when content is read.

When reading content, an offset must be specified. This is the number of octets from the first octet belonging to the basic entity. Other parameters of the functions specify the length of the buffer and data in octets. This means that content can be manipulated using buffers of any size. The functions are as follows:

dla_get_content
dla_get_content_length
dla_get_spec_content
dla_get_spec_content_length

For an example, see 2.2.12.3, which covers downward navigation.

### 2.2.4 Toolkit and Document Management Operations for Creating Documents

The following code fragments create documents whose handles are used in the examples in the rest of 2.2. The handles are:

- document_hdl, DAP level 2 document
- document2_hdl, DAP level 3 document
- gen_doc_hdl, generic document
- with_ext_doc_hdl, DAP level 3 document that uses an external document
- ext_doc_hdl, external document
- with_res_doc_hdl, DAP level 3 document that uses a resource document
- res_doc_hdl, resource document

#### 2.2.4.1 Creating Ordinary Documents and Using Modes

This example is a general application skeleton. It illustrates:

- Initializing the Toolkit
- Creating new documents in mode-1 and mode-2
- Retrieving the DocumentProfile to set some of its properties
- Writing documents

- Investigating errors
- Terminating the Toolkit

```
main()
{
dla_status                      return_status;
dla_toolkit_handle              toolkit_hdl;
dla_document_handle             document_hdl;
dla_document_handle             document2_hdl;
dla_entity_handle               doc_prof_hdl;
dla_dap_level                   dap_level;
dla_entity_handle               entity_handle_return;
dla_property_code               property_code_return;
dla_integer                     err_num_return;
ola_constituent_handle          constituent_handle_return;
ola_attribute_code              attribute_code_return;
ola_integer                     ola_err_num_return;

return_status =
    dla_init_toolkit(
          DLA_C_NO_OPTIONS,
          0,
          0,
          &toolkit_hdl);

dap_level = DLA_C_DAP_LEVEL_2;
return_status =
    dla_create_document(
          toolkit_hdl,
          dap_level,
          DLA_C_NO_OPTIONS,
          DLA_C_NULL_STRING,
          0,
          &document_hdl);

/*
 * Create a document at DAP Level 3 using mode-2, i.e.
 * application creates the generic logical structure
 *
 * This document handle will be used in later examples
 */
dap_level = DLA_C_DAP_LEVEL_3;
return_status =
    dla_create_document(
    toolkit_hdl,
    dap_level,
    DLA_C_MODE_2,
    DLA_C_NULL_STRING,
    0,
    &document2_hdl);

return_status =
    dla_find_profile(
          document_hdl,
          &doc_prof_hdl);
/*
 * Properties may be set on the DocumentProfile and the rest
 * of the document processing performed.
 */

/*
 * Now write the document
 *
 */
```

```
return_status =
    dla_write_document(
          document_hdl,
          "filename.odif",
          strlen("filename.odif"),
          DLA_C_NO_OPTIONS);

if (      dla_failed(return_status) &&
          dla_error_class(return_status) ==
                    DLA_C_CONSTRUCT_ERROR )
{
    return_status =
          dla_get_construct_error(
                document_hdl,
                &entity_handle_return,
                &property_code_return,
                &dla_err_num_return);

          /* code to display or capture the reason for failure */
}

else
if (      dla_failed(return_status) &&
          dla_error_number(return_status) ==
                    DLA_E_OLA_WRITE_ERROR )
{
    return_status =
          dla_get_ola_write_error(
                document_hdl,
                &constituent_handle_return,
                &attribute_code_return,
                &ola_err_num_return);

                /* code to display or capture the
                 * reason for failure
                 */
}

return_status =
    dla_term_toolkit(toolkit_hdl);

}
```

### 2.2.4.2 Creating a Document using Generic Documents

A generic document can be either a resource document or an external document. This subclause demonstrates the creation of a document using an external document or a resource document.

```
dla_status                  return_status;
dla_toolkit_handle          toolkit_hdl;
dla_dap_level               dap_level;
dla_document_handle         with_ext_doc_hdl;
dla_document_handle         ext_doc_hdl;
dla_document_handle         with_res_doc_hdl;
dla_document_handle         res_doc_hdl;
```

```
/*
 * Create a document using an external document
 */
dap_level = DLA_C_DAP_LEVEL_3;
return_status =
    dla_create_document(
    toolkit_hdl,
    dap_level,
    DLA_C_WITH_EXTERNAL,
    DLA_C_NULL_STRING,
    0,
    &with_ext_doc_hdl);

/*
 * Read the generic document as an external document
 */
return_status =
    dla_read_generic_doc(with_ext_doc_hdl,
                         "extfile.odif",
                         strlen("extfile.odif"),
                         DLA_C_EXTERNAL_DOC,
                         &ext_doc_hdl);

/*
 * Create a document using a resource document
 */
dap_level = DLA_C_DAP_LEVEL_3;
return_status =
    dla_create_document(
    toolkit_hdl,
    dap_level,
    DLA_C_MODE_2,
    DLA_C_NULL_STRING,
    0,
    &with_res_doc_hdl);
/*
 * Read the generic document as a resource document
 */
return_status =
    dla_read_generic_doc(with_res_doc_hdl,
                         "resfile.odif",
                         strlen("resfile.odif"),
                         DLA_C_RESOURCE_DOC,
                         &res_doc_hdl);
```

#### 2.2.4.3 Creating Generic Documents

This subclause demonstrates the creation of a generic document that can be used as an external document.

```
dla_status                  return_status;
dla_toolkit_handle          toolkit_handle;
dla_dap_level               dap_level;
dla_document_handle         gen_doc_hdl;
```

```
/*
 * Create a document which will be used as a generic document.
 */
dap_level = DLA_C_DAP_LEVEL_3;
return_status =
    dla_create_document(
    toolkit_hdl,
    dap_level,
    DLA_C_FOR_EXTERNAL,
    DLA_C_NULL_STRING,
    0,
    &gen_doc_hdl);
```

The application must now create the generic logical structure and the generic layout structure.

Before the document can be used as a generic document, the function dla_write_document must be called to generate ODIF and the function dla_read_generic_doc must be called to read the document as an external document. Resource documents are created and used in a similar way.

### 2.2.5 Creating Generic Logical Entities

This subclause demonstrates the creation of generic logical entities and the use of a resource document. It also demonstrates the explicit assignment of styles to a generic logical entity. The explicit assignment of styles to a specific logical entity is done in the same way.

```
dla_status                  return_status;
dla_document_handle         document2_hdl;
dla_entity_handle           gen_log_root_hdl;
dla_entity_handle           gen_passage1_hdl;
dla_entity_handle           gen_passage2_hdl;
dla_entity_handle           gen_title_hdl;
dla_entity_handle           lay_style_hdl;
dla_document_handle         with_res_doc_hdl;
dla_entity_handle           gen_para_hdl;
dla_entity_handle           res_body_ras_hdl;
dla_entity_handle           gen_body_ras_hdl;
```

This code fragment demonstrates the creation of generic logical entities without a generic document.

```
/*
 * Create the generic logical root
 */
return_status =
    dla_create_gen_log_entity(
          document2_hdl,
          DLA_C_GEN_LOG_ROOT,
          DLA_C_NULL_HANDLE,
          &gen_log_root_hdl);
/*
 * Create two generic Passages
 */
return_status =
    dla_create_gen_log_entity(
          document2_hdl,
          DLA_C_GEN_PASSAGE,
          gen_log_root_hdl,
          &gen_passage1_hdl);
```

```
return_status =
    dla_create_gen_log_entity(
          document2_hdl,
          DLA_C_GEN_PASSAGE,
          gen_log_root_hdl,
          &gen_passage2_hdl);
/*
 * Create a generic Title
 */
return_status =
    dla_create_gen_log_entity(
          document2_hdl,
          DLA_C_GEN_TITLE,
          gen_passage1_hdl,
          &gen_title_hdl);

/*
 * Set a Layout Directive associated with a generic Passage
 */
return_status =
    dla_set_integer(
          document2_hdl,
          gen_passage1_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_LYD_IDV_PGE,
          DLA_C_LAY_TY_PGE);

/*
 * Assign the style associated with a generic Passage with
 * another generic Passage.
 *
 * As generic Passage and generic Title have LStyle2 layout
 * styles in FOD36 the style can also be assigned to the
 * generic Title, gen_title_hdl
 */
return_status =
    dla_get_spec_entity_handle(
          document2_hdl,
          gen_passage1_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_LAY_STYLE,
          &lay_style_hdl,
          &property_status);

return_status =
    dla_set_entity_handle(
          document2_hdl,
          gen_passage2_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_LAY_STYLE,
          lay_style_hdl);
```

The following code fragment demonstrates the use of a resource document when creating a generic logical entity.

Assume that the handle of a generic BodyRaster entity, res_body_ras_hdl, has been found in the Resource Document, and that a generic paragraph, gen_para_hdl, has been created.

```
/*
 * Create a generic BodyRaster
 */
return_status =
    dla_create_gen_log_entity(
           with_res_doc_hdl,
           DLA_C_GEN_BODY_RASTER,
           gen_para_hdl,
           &gen_body_ras_hdl);
/*
 * Specify that res_body_ras_hdl is to be used as a Resource
 * gen_body_ras_hdl
 */
return_status =
    dla_set_entity_handle(
           with_res_doc_hdl,
           gen_body_ras_hdl,
           DLA_C_NULL_HANDLE,
           DLA_C_RESOURCE,
           res_body_ras_hdl);
```

### 2.2.6 Creating Specific Logical Entities

This subclause demonstrates the creation of specific logical entities using mode-1 and mode-2, including using an external document. It also demonstrates the use of the command dla_unfix_properties.

The explicit assignment of styles to specific logical entities is exactly the same as for generic logical entities.

#### 1.2.6.1 Creating Specific Logical Entities Using Mode-1

Figure 6 shows part of the specific logical structure of a document.

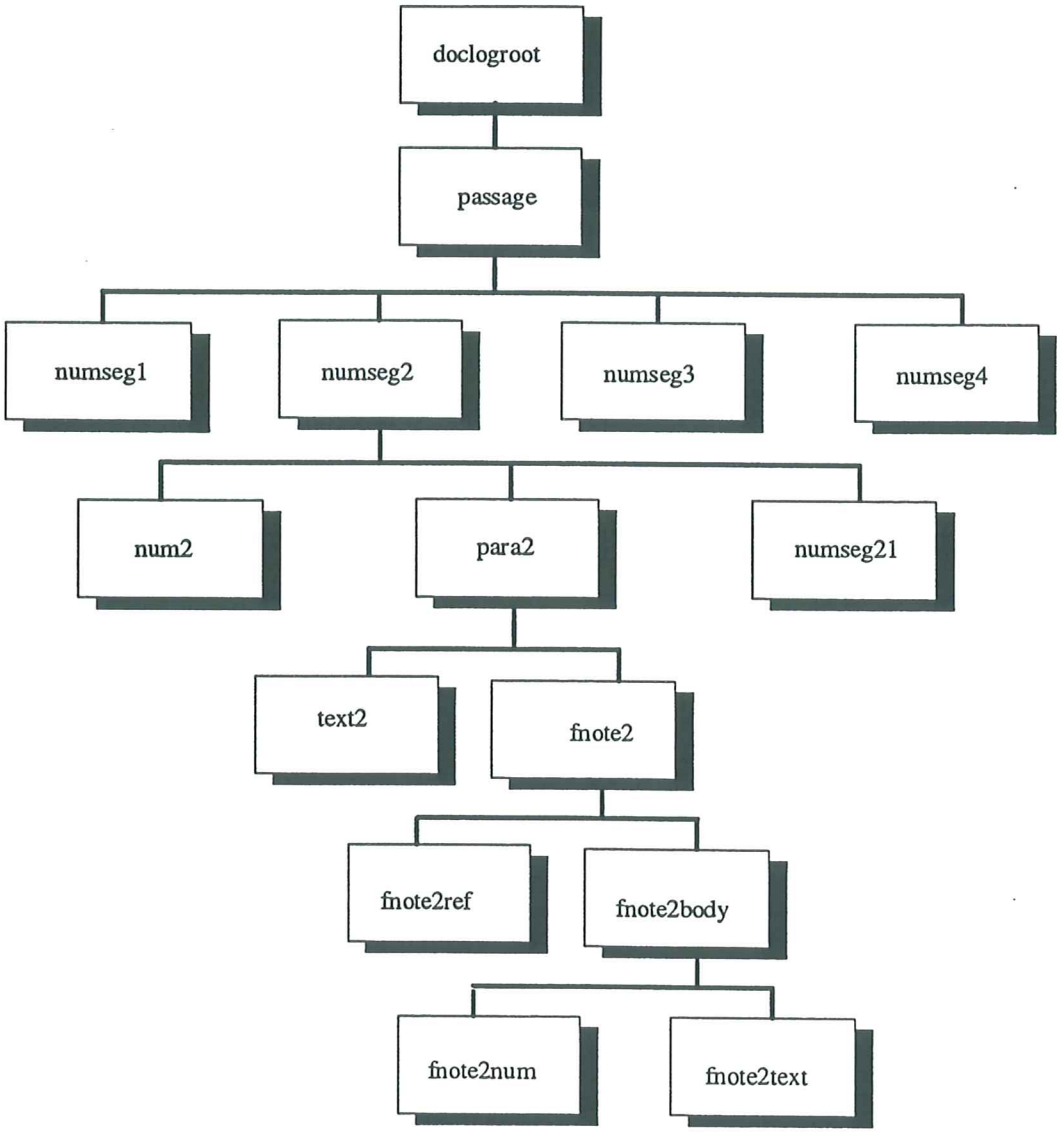



**Figure 6 - Example of Specific Logical Structure**

This subclause demonstrates the creation of the structure shown in Figure 6. It illustrates the following:

- Retrieving the handle of specific DocumentLogicalRoot
- Creating an entity and linking it to its parent
- Creating an entity using the function dla_duplicate_specific_entity
- Creating an entity cluster
- Retrieving an entity within a cluster

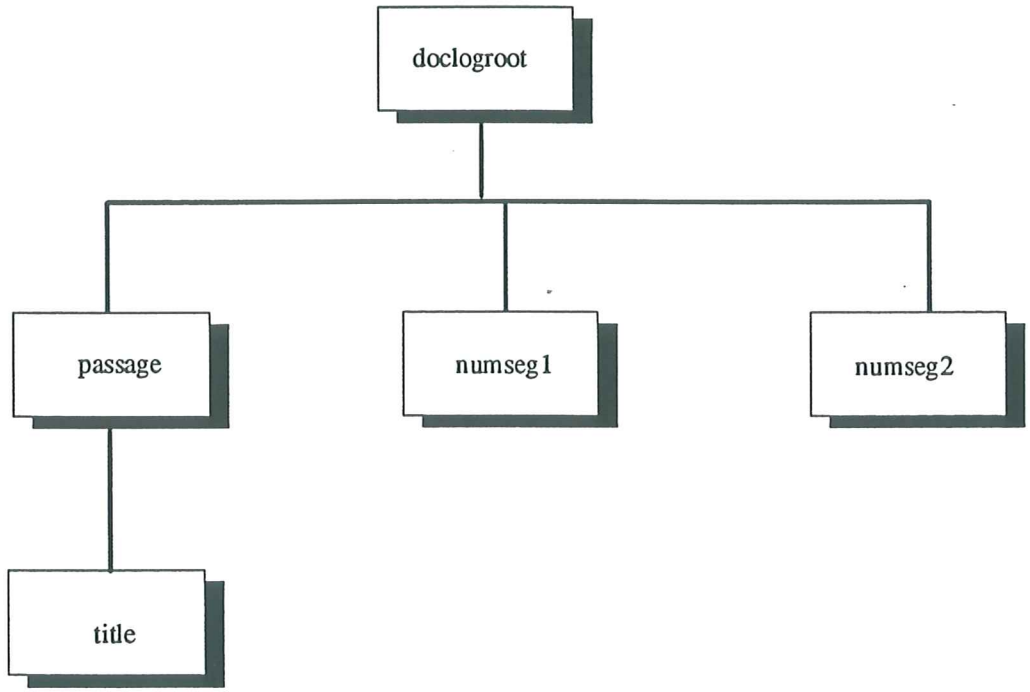
```
dla_status                  return_status;
dla_document_handle         document_hdl;
dla_entity_handle           doclogroot_hdl;
dla_entity_handle           passage_hdl;
dla_entity_handle           numseg1_hdl;
dla_entity_handle           numseg2_hdl;
dla_entity_handle           para2_hdl;
dla_entity_handle           text2_hdl;
dla_entity_handle           fnote2_hdl;
dla_entity_handle           numseg21_hdl;
dla_entity_handle           fnote2body_hdl;
dla_property_status         property_status;
dla_derived_location        derived_location;
dla_entity_handle           derived_entity_hdl;
dla_entity_handle           fnote2text_hdl;

/*
 * Retrieve the handle of the specific DocumentLogicalRoot.
 */
return_status =
    dla_move_to_root_entity(
          document_hdl,
          DLA_C_SPECIFIC_LOGICAL,
          &doclogroot_hdl);

/*
 * Create a passage entity and link it to the specific
 * DocumentLogicalRoot.
 */

return_status =
    dla_create_specific_entity(
          document_hdl,
          DLA_C_SPEC_PASSAGE,
          doclogroot_hdl,
          DLA_C_LINK_AS_FIRST,
          &passage_hdl);

/*

 * Create the Numbered Segments required and link them to
 * the Passage.
 */

return_status =
    dla_create_specific_entity(
          document_hdl,
          DLA_C_SPEC_NUM_SEG,
          passage_hdl,
          DLA_C_LINK_AS_LAST,
          &numseg1_hdl);

/*
 * Note that the call to dla_fix_properties is required for
 * entity before it can be used as a model entity in the
 * function dla_duplicate_specific_entity. The numbering format
 * is assumed to have been established.
 *
 * The creation of a Paragraph after numseg1
 * (using the link option DLA_C_LINK_AFTER) is not permitted.
 */
```

```
return_status =
    dla_fix_properties(
          document_hdl,
          numseg1_hdl);

return_status =
    dla_duplicate_specific_entity(
          document_hdl,
          numseg1_hdl, /* model entity */
          passage_hdl, /* link entity */
          DLA_C_LINK_AS_LAST,
          &num_seg2hdl);

/*
 * The Numbered Segments numseg3 and numseg4 can be created
 * in exactly the same way.
 */

/*
 * Create the entities para2, text2, fnote2 and numseg21.
 *
 * The entity num2 has been created automatically by the
 * creation of the entity numseg2.
 */

return_status =
    dla_create_specific_entity(
          document_hdl,
          DLA_C_SPEC_PARAGRAPH,
          numseg2_hdl,
          DLA_C_LINK_AS_LAST,
          &para2_hdl);

return_status =
    dla_create_specific_entity(
          document_hdl,
          DLA_C_SPEC_BODY_TEXT,
          para2_hdl,
          DLA_C_LINK_AS_LAST,
          &text2_hdl);

return_status =
    dla_create_specific_entity(
          document_hdl,
          DLA_C_SPEC_FOOTNOTE,
          para2_hdl,
          DLA_C_LINK_AS_LAST,
          &fnote2_hdl);

return_status =
    dla_create_specific_entity(
          document_hdl,
          DLA_C_SPEC_NUM_SEG,
          numseg2_hdl,
          DLA_C_LINK_AS_LAST,
          &numseg21_hdl);

/*
 * Retrieve the handle of the entity fnote2body which was
 * created automatically by the creation of the entity
 * fnote2.
 */
```

```
return_status =
    dla_get_spec_entity_handle(
            document_hdl,
            fnote2_hdl,
            DLA_C_NULL_HANDLE,
            DLA_C_ENT_FNOTE_BODY,
            &fnote2body_hdl,
            &property_status);

return_status =
    dla_create_specific_entity(
            document_hdl,
            DLA_C_SPEC_FOOTNOTE_TEXT,
            fnote2body_hdl,
            DLA_C_LINK_AS_LAST,
            &fnote2text_hdl);
```

#### 2.2.6.2 Creating Specific Logical Entities Using Mode-2

Figure 7 shows part of the specific logical structure of a document.




ECMA-93-0009-A

**Figure 7 - Example of Specific Logical Structure**

This subclause demonstrates the creation of the structure shown in Figure 7. The use, in mode-2, of the following functions is illustrated:

- dla_instantiate_spec_log
- dla_duplicate_specific_entity

The creation of some of the generic logical entities that are required is shown in the example in 2.2.5.

```
dla_status                  return_status;
dla_document_handle         document2_hdl;
dla_entity_handle           gen_log_root_hdl;
dla_entity_handle           doclogroot;
dla_entity_handle           gen_passage1_hdl;
dla_entity_handle           passage;
dla_entity_handle           gen_title_hdl;
```

```
dla_entity_handle               title;
dla_entity_handle               gen_num_seg_hdl;
dla_entity_handle               numseg1;
dla_entity_handle               numseg2;

/*
 * Create the specific DocumentLogicalRoot
 */
return_status =
    dla_instantiate_spec_log(
           document2_hdl,
           gen_log_root_hdl,
           DLA_C_NULL_HANDLE,
           DLA_C_LINK_AS_FIRST,
           &doclogroot);

/*
 * Create the specific Passage
 */
return_status =
    dla_instantiate_spec_log(
           document2_hdl,
           gen_passage1_hdl,
           doclogroot,
           DLA_C_LINK_AS_FIRST,
           &passage);

 *
 * Create the specific Title
 */
    return_status =
    dla_instantiate_spec_log(
           document2_hdl,
           gen_title_hdl,
           passage,
           DLA_C_LINK_AS_LAST,
           &title);

/*
 * Create the generic NumberedSegment
 */
return_status =
    dla_create_gen_log_entity(
           document2_hdl,
           DLA_C_GEN_NUM_SEG,
           gen_log_root_hdl,
           &gen_num_seg_hdl);

/*
 * Create one specific NumberedSegment
 */
return_status =
    dla_instantiate_spec_log(
           document2_hdl,
           gen_num_seg_hdl,
           passage,
           DLA_C_LINK_AFTER,
           &numseg1);
```

```
/*
 * Call dla_fix_properties before calling
 * dla_duplicate_specific_entity
 */
return_status =
    dla_fix_properties(
           document2_hdl,
           numseg1);

/*
 * Create the other specific NumberedSegment
 */
return_status =
    dla_duplicate_specific_entity(
           document2_hdl,
           numseg1,
           doclogroot,
           DLA_C_LINK_AS_LAST,
           &numseg2);
```

### 2.2.6.3 Creating Specific Logical Entities Using an External Document

This subclause illustrates the creation of a specific DocumentLogicalRoot using the generic DocumentLogicalRoot in an external document.

```
dla_status                  return_status;
dla_document_handle         with_ext_doc_hdl;
dla_document_handle         ext_doc_hdl;
dla_entity_handle           gen_log_root_hdl;
dla_entity_handle           doclogroot;
dla_entity_type             entity_type;

/*
 * Get the handle of the generic DocumentLogicalRoot
 */
return_status =
    dla_move_to_root_entity(
           ext_doc_hdl,
           DLA_C_GENERIC_LOGICAL,
           &gen_log_root_hdl);
/*
 * Alternatively
 */
return_status =
    dla_locate_gen_entity(
           with_ext_doc_hdl,
           "2",
           1,
           &gen_log_root_hdl,
           &entity_type);      /* Should be DLA_C_GEN_LOG_ROOT */
/*
 * Now create the specific DocumentLogicalRoot
 */
return_status =
    dla_instantiate_spec_log(
           with_ext_doc_hdl,
           gen_log_root_hdl,
           DLA_C_NULL_HANDLE,
           DLA_C_LINK_AS_FIRST,
           &doclogroot);
```

#### 2.2.6.4 Using dla_unfix_properties

This subclause illustrates the use of dla_unfix_properties. Although the example is for a document created using mode-2, the function applies to a document created using any mode.

```
dla_status                  return_status;
dla_document_handle         document2_hdl;
dla_entity_handle           numseg1;

/*
 * Call dla_unfix_properties
 */
return_status =
    dla_unfix_properties(
          document2_hdl,
          numseg1);
```

Properties can now be set on the specific logical NumberedSegment numseg1.

### 2.2.7 Creating Generic Layout Entities

This subclause illustrates the creation of entities in the generic layout structure. It includes examples of the following:

- Creating a generic Pageset
- Creating a Header and its SourcedContentVariable frames on the Title Page of a document
- Creating a CommonText entity associated with a SourcedContentFixed frame
- Creating a VariableCompositeBody frame on a RectoPage
- Linking a previously created VariableCompositeBody frame with a VersoPage
- Creating a Footnote Area in a VariableCompositeBody frame
- Creating a BasicFloat frame
- Creating a SnakingColumn frame and its ColumnVariable columns
- Creating a SychronizedColumn frame and its ColumnFixed columns
- Creating a generic Table Area

```
dla_status                  status;
dla_document_handle         document_hdl;
dla_pageset_type            pageset_type;
dla_entity_handle           pageset_hdl;
dla_entity_handle           title_page_hdl;
dla_entity_handle           recto_page_hdl;
dla_entity_handle           verso_page_hdl;
dla_property_status         property_status;
dla_derived_location        derived_location;
dla_entity_handle           derived_entity_hdl;
dla_integer                 number_of_frames;
dla_entity_handle           header_hdl;
dla_entity_handle           srcd_cont_var_hdl;
dla_entity_handle           common_text_hdl;
dla_entity_handle           body_hdl;
dla_entity_handle           footnote_area_hdl;
dla_entity_handle           basic_float_hdl;
```

#### 2.2.7.1 Creating a Generic Pageset

This example creates a generic Pageset that consists of a Title Page and a pair of Recto and Verso Pages.

```
pageset_type = DLA_C_TITLE_VERSO_RECTO;

return_status =
    dla_create_generic_pageset(
          document_hdl,
          pageset_type,
          &pageset_hdl);
```

#### 2.2.7.2 Creating a Generic Header

This example creates a generic Header frame on a Title Page. It consists of the following steps:

- Retrieving the handle of the Title Page
- Creating the Header frame and its subordinate frames that are of type SourcedContentVariable
- Retrieving the handle of one of the SourcedContentVariable frames
- Creating a CommonText entity associated with the SourcedContentVariable frame

```
/* Retrieve the handle of the Title Page */

return_status =
    dla_get_spec_entity_handle(
           document_hdl,
           pageset_hdl,
           DLA_C_NULL_HANDLE,
           DLA_C_GEN_INIT_PGE,
           &title_page_hdl);

/*
 * Create the header area and the subordinate frames.
 * The document is at DAP level 2.
 */

number_of_frames = 2;
return_status =
    dla_create_generic_head_footer(
           document_hdl,
           title_page_hdl,
           DLA_C_COMP_HEAD_VAR,
           number_of_frames,
           &header_hdl);

/*
 * Get the handle of the second SourceContentVariable frame
 * created by the previous call.
 */

return_status =
    dla_get_spec_index_entity(
           document_hdl,
           header_hdl,
           DLA_C_NULL_HANDLE,
           DLA_C_HED_FOT_FRM,
           2,
           &srcd_cont_var_hdl,
           &property_status);

/*
 * Create a CommonText entity associated with this
 * SourcedContentVariable frame.
 */

return_status =
    dla_create_generic_comm_entity(
           document_hdl,
           srcd_cont_var_hdl,
           DLA_C_GEN_COMMON_TEXT,
           &common_text_hdl);
```

#### 2.2.7.3 Creating a Generic VariableCompositeBody Frame

This example creates a generic VariableCompositeBody frame subordinate to a Recto Page. It consists of the following steps:

- Retrieving the handle of the Recto Page
- Creating the VariableCompositeBody frame

```
/* Retrieve the handle of the Recto Page */

return_status =
    dla_get_spec_entity_handle(
          document_hdl,
          pageset_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_GEN_REC_PGE,
          &recto_page_hdl,
          &property_status);

/* Now create the VariableCompositeBody frame */

return_status =

    dla_create_generic_body_frame(
          document_hdl,
          recto_page_hdl,
          DLA_C_GEN_COMP_BODY_VAR,
          &body_hdl);
```

#### 2.2.7.4 Linking Entities in the Generic Structure

This example links a previously created VariableCompositeBody frame with a Verso Page. It consists of the following steps:

- Retrieving the handle of the Verso Page
- Linking the VariableCompositeBody frame with the Verso Page

```
/*
 *
 * Retrieve the handle of the Verso Page
 */

return_status =
    dla_get_spec_entity_handle(
          document_hdl,
          pageset_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_GEN_VER_PGE,
          &verso_page_hdl,
          &property_status);

/*
 * Now link the VariableCompositeBody frame with the
 * Verso Page.
 */

return_status =
    dla_link_generic_entity(
          document_hdl,
          verso_page_hdl,
          body_hdl);
```

#### 2.2.7.5 Creating a Generic Footnote Area

This example creates a generic Footnote Area in a VariableCompositeBody frame.

```
return_status =
    dla_create_generic_fnote_area(
           document_hdl,
           body_frame_hdl,
           &footnote_area_hdl);
```

#### 2.2.7.6 Creating a Generic BasicFloat Frame

This example creates a generic BasicFloat frame in a VariableCompositeBody frame.

```
return_status =
    dla_create_generic_basic_float(
           document_hdl,
           body_frame_hdl,
           &basic_float_hdl);
```

#### 2.2.7.7 Creating a Generic SnakingColumn Frame

This example illustrates the creation of a generic SnakingColumn frame and its subordinate ColumnVariable frames, and the retrieval of one of the ColumnVariable frames in order to set some of its properties.

The application must specify which type of SnakingColumn frame is required, in this example it is DLA_C_SNAKING_SEQ. This type requires that the number of ColumnVariable frames be specified.

```
dla_snaking_type            snaking_type;
dla_integer                 number_of_columns;
dla_entity_handle           snaking_col_hdl;
dla_entity_handle           col_var_hdl;

snaking_type = DLA_C_SNAKING_SEQ;
number_of_columns = 4;
return_status =
    dla_create_generic_snaking(
           document_hdl,
           body_hdl,
           snaking_type,
           number_of_columns,
           &snaking_col_hdl);

/*
 * Now retrieve the first ColumnVariable frame in order
 * to set some of its properties.
 */

return_status
    dla_get_spec_index_entity(
           document_hdl,
           snaking_col_hdl,
           DLA_C_NULL_HANDLE,
           DLA_C_ONE_COL_VAR,
           1,
           &col_var_hdl,
           &property_status);
```

### 2.2.7.8 Creating a Generic SynchronizedColumn Frame

This example illustrates the creation of a generic SynchronizedColumn frame and its subordinate ColumnFixed frames.

The application must specify the number of ColumnFixed frames that are required.

```
dla_entity_handle       sync_col_hdl;
dla_entity_handle       col_fix_hdl;

number_of_columns = 2;
return_status =
    dla_create_generic_synchronized(
          document_hdl,
          body_hdl,
          number_of_columns,
          &sync_col_hdl);
```

### 2.2.7.9 Creating a Generic TableArea

This subclause illustrates the creation of a generic TableArea.

This entity can only be created at DAP level 3; it is assumed that a generic VariableCompositeBody has already been created with the handle body_hdl.

This example illustrates the use of the following functions:

- dla_create_generic_frame
- dla_create_generic_frame_seq
- dla_link_generic_entity
- dla_link_generic_entity_seq

```
dla_status                  return_status;
dla_document_handle         document2_hdl;
dla_entity_handle           body_area_hdl;
dla_entity_handle           table_area_hdl;
dla_entity_handle           table_label_hdl;
dla_entity_handle           table_label_cont_hdl;
dla_entity_handle           comp_table_label_hdl;
dla_entity_handle           label_comp_hdl;
dla_entity_handle           row_area_hdl;
dla_entity_handle           cell1_hdl;
dla_entity_handle           cell2_hdl;
dla_entity_handle           sub_row_grp_hdl;
dla_entity_handle           sub_row_hdl;

/*
 * Create a generic TableArea and append it to the list of
 * entities in the GFS of body_hdl
 */
return_status =
    dla_create_generic_frame(
          document2_hdl,
          body_hdl,
          DLA_C_GEN_TABLE_AREA,
          &table_area_hdl);
```

```
/*
 * Create a generic TableLabel
 */
return_status =
    dla_create_generic_frame(
           document2_hdl,
           table_area_hdl,
           DLA_C_GEN_TABLE_LABEL,
           &table_label_hdl);

/*
 * Create a generic TableLabelContent
 */
return_status =
    dla_create_generic_frame(
           document2_hdl,
           table_label_hdl,
           DLA_C_GEN_TABLE_LABEL_CONTENT,
           &table_label_cont_hdl);

/*
 * Create a generic CompositeTableLabel
 */
return_status =
    dla_create_generic_frame(
           document2_hdl,
           table_label_hdl,
           DLA_C_GEN_COMP_TABLE_LABEL,
           &comp_table_label_hdl);

/*
 * Create a generic LabelComponent
 */
return_status =
    dla_create_generic_frame(
           document2_hdl,
           comp_table_label_hdl,
           DLA_C_GEN_LABEL_COMPONENT,
           &label_comp_hdl);

/*
 * Link a previously created generic TableLabelContent
 */
return_status =
    dla_link_generic_entity(
           document2_hdl,
           label_comp_hdl,
           table_label_cont_hdl);

/*
 * The creation of the TableLabel is now complete.
 *
 * Create a generic RowArea
 */
return_status =
    dla_create_generic_frame_seq(
           document2_hdl,
           table_area_hdl,
           DLA_C_GEN_ROW_AREA,
           2,
           &row_area_hdl);
```

```
/*
 * Create a generic Cell
 */
return_status =
    dla_create_generic_frame(
           document2_hdl,
           row_area_hdl,
           DLA_C_GEN_CELL,
           &cell1_hdl);

/*
 * Create a generic SubRowGroup
 */
return_status =
    dla_create_generic_frame(
           document2_hdl,
           row_area_hdl,
           DLA_C_GEN_SUBROW_GROUP,
           &sub_row_grp_hdl);
/*
 * Create a generic SubRow
 */
return_status =
    dla_create_generic_frame(
           document2_hdl,
           sub_row_grp_hdl,
           DLA_C_GEN_SUBROW,
           &sub_row_hdl);
/*
 *Create another generic Cell
/*
return_status =
    dla_create_generic_frame(
           document2_hdl,
           sub_row_hdl,
           DLA_C_GEN_CELL,
           &cell2_hdl);
/*
 * Link a previously created generic Cell
 */
return_status =
    dla_link_generic_entity_seq(
           document2_hdl,
           sub_row_hdl,
           1,
           cell1_hdl);
```

### 2.2.8 Property Setting

This subclause illustrates the setting of properties of DAP entities. It contains a representative sample of DAP properties and uses most of the functions provided by the DAP API to set properties.

It demonstrates the setting of the following properties:

- User visible name on a Numbered Segment
- Initial value for the number of the first Numbered Segment, and Separator string for all Numbered Segments, at level 2 in a Passage
- Table numbering for a generic TableNumber
- Page break for a Numbered Segment
- Column break for a Numbered Segment
- Page layout type for a Recto Page
- Path dimension for a ColumnVariable frame

- Tabulation stop
- Graphic character set

#### 2.2.8.1 Set User Visible Name

This example illustrates the setting of the value of the property User visible name on a Numbered Segment.

```
dla_integer                    string_length;
char *                         string_name;

string_name = "abc";
string_length = 3;
return_status =
    dla_set_string(
          document_hdl,
          numseg1_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_USR_NAM,
          (dla_octet_string)string_name,
          string_length);
```

#### 2.2.8.2 Set Segment Number Initialization

This example illustrates the setting of an Initial value and Separator string for a Numbered Segment. These properties are generic entity properties and so can only be set through specific entities in mode-1 documents. The example would be identical for generic entities in other modes.

It is assumed that the requirement is to set the Initial value and Separator string for all Numbered Segments at level 2 in a Passage.

```
dla_structure_handle           level2_init_hdl;
dla_integer                    separator_string_len;
char *                         separator_string;

/*
 * Create the structure to hold the values as a property
 * of the passage.
 */

return_status =
    dla_set_index_structure(
          document_hdl,
          passage_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_SEG_NUM_INIT,
          1,
          DLA_C_INDEX_INSERT,
          &level2_init_hdl);

/*
 * Specify that the structure will be used to hold Segment
 * Number Initialization values for Numbered Segments at
 * level 2.
 */

return_status =
    dla_set_integer(
          document_hdl,
          passage_hdl,
          level2_init_hdl,
          DLA_C_SEG_NUM_LEV_NUM,
          2);
```

```
/*
 * Now set the Initial Value to 1; i.e. the value for the
 * number of the first numbered segment at level 2.
 */
return_status =
    dla_set_integer(
           document_hdl,
           passage_hdl,
           level2_init_hdl,
           DLA_C_SEG_NUM_INIT_VAL,
           1);

/* Now specify the separator string. */

separator_string = ".";
separator_string_length = 1;
return_status =
    dla_set_string(
           document_hdl,
           passage_hdl,
           level2_init_hdl,
           DLA_C_SEG_NUM_SEP_STR,
           (dla_octet_string)separator_string,
           separator_string_len);
```

#### 2.2.8.3 Set TableNumbers Numbering Property

This example specifies the mandatory <num-str> and optional <pre-str> of the macro TableNumbers on a generic TableNumber. For additional relevant information, see 7.3.1 of FOD36 (the information is approximately in the middle of this rather long section).

It is assumed that the document handle is document2_hdl and that a generic TableNumber with handle gen_tab_num_hdl and a generic Table with handle gen_tab_hdl have already been created.

```
/*
 * Specify the TableNumbers class
 */
return_status =
    dla_set_entity_handle(
           document2_hdl,
           gen_tab_num_hdl,
           DLA_C_NULL_HANDLE,
           DLA_C_TAB_NUM_NBS_CLS,
           gen_tab_hdl);

/*
 * Specify the TableNumbers numString Index
 */
return_status =
    dla_set_integer(
           document2_hdl,
           gen_tab_num_hdl,
           DLA_C_NULL_HANDLE,
           DLA_C_TAB_NUM_NBS_IDX,
           1);
```

```
/*
 * Specify the TableNumbers Prefix String Index
 */
return_status =
    dla_set_integer(
           document2_hdl,
           gen_tab_num_hdl,
           DLA_C_NULL_HANDLE,
           DLA_C_TAB_NUM_PRE_IDX,
           1);
```

#### 2.2.8.4 Set Page Break

This example specifies the property Page break for a Numbered Segment, so that the Numbered Segment is laid out starting on the next available page.

```
return_status =
    dla_set_integer(
           document_hdl,
           numseg1_hdl,
           DLA_C_NULL_HANDLE,
           DLA_C_LYD_PGE_BRK,
           DLA_C_LAY_TY_PGE);
```

#### 2.2.8.5 Set Column Break

This example specifies the property Column break for a Numbered Segment.

The type of the entity whose handle is specified must be (for a FOD26 document) DLA_C_GEN_SNAKING_COLUMN. If the Page break property has been set previously, the setting of this property supersedes the Page break property.

```
return_status =
    dla_set_entity_handle(
           document_hdl,
           numseg1_hdl,
           DLA_C_NULL_HANDLE,
           DLA_C_LYD_COL_BRK,
           snaking_col_hdl);
```

#### 2.2.8.6 Set Page Layout Type

This example sets the Page layout type for a Recto Page to page layout type A. This also sets the Page layout type for the twinned Verso Page.

```
return_status =
    dla_set_integer(
           document_hdl,
           recto_page_hdl,
           DLA_C_NULL_HANDLE,
           DLA_C_PGE_LAY_TYP,
           DLA_C_PGE_LAY_A);
```

#### 2.2.8.7 Set Path Dimension

This example sets the Path dimension for a ColumnVariable frame to 1000 BMUs.

The Page layout type must have been set for the Generic Page that owns the ColumnVariable frame.

```
return_status =
    dla_set_integer(
           document_hdl,
           col_var_hdl,
           DLA_C_NULL_HANDLE,
           DLA_C_PTH_DMS_FIX,
           1000);
```

#### 2.2.8.8 Set Tabulation Stop

This example sets a tabulation stop for the entity text2.

```
dla_octet_string              tab_reference;
dla_integer                   tab_reference_len;

/*
 * Assume that memory has been allocated for the tab
 * reference string and that its address has been set up in
 * the variable tab_reference and its length in the variable
 * tab_reference_length.
 */

return_status =
    dla_set_tab_stop(
          document_hdl,
          text2_hdl,
          tab_reference,
          tab_reference_len,
          100,                    /* tab position in BMUs */
          DLA_C_TAB_ALIGN_START,
          DLA_C_NULL_STRING,      /* No Align around string */
          0);
```

#### 2.2.8.9 Set Graphic Character Set

This example designates a 96-character single-byte character set for G1.

```
dla_octet_string              last_char_id;

/*
 * Allocate one byte of memory for that octet string and
 * copy the last character identifier into the first (only)
 * byte of this memory.
 */

return_status =
    dla_append_char_set(
          document_hdl,
          text_hdl,
          DLA_C_CA_GR_CHR_SET,
          last_char_id,
          1,
          DLA_C_SIN_96,
          DLA_C_G1);
```

### 2.2.9 Adding Content

This subclause demonstrates the following:

- Adding content for a header
- Adding more content for a header

#### 2.2.9.1 Add Content for a Header

This example adds content to the CommonText entity created in 2.2.7.2.

```
dla_octet_string              data_buffer;
dla_integer                   buffer_size;

/*
 * Allocate storage for data buffer, place the data in
 * the buffer and set the length of the data.
 */
/*
```

```
 * Now add the data in the buffer to the content associated with
 * the CommonText entity.
 */
return_status =
    dla_add_content(
           document_hdl,
           common_text_hdl,
           0,
           buffer_size,
           data_buffer);
```

#### 2.2.9.2 Add More Content for a Header

This example illustrates the addition of more content to the CommonText entity to which content was added in 2.2.9.1. It consists of the following steps:

- Retrieving the current content length
- Adding the extra content

```
dla_length                  content_length;
dla_octet_string            data_buffer;
dla_integer                 buffer_size;

/* Get the current content length */

return_status =
    dla_get_content_length(
           document_hdl,
           common_text_hdl,
           &content_length);

/*
 * Now allocate storage for data buffer, place the data in
 * the buffer and set the length of the data.
 *
 * Add the data in the buffer to the content associated with
 * the CommonText entity.
 */

return_status =
    dla_add_content(
           document_hdl,
           common_text_hdl,
           content_length,
           buffer_size,
           data_buffer);
```

### 2.2.10 Toolkit and Document Management Operations for Reading Documents

This example is a general application skeleton. It illustrates:

- Initializing the Toolkit
- Reading an existing document
- Reading a generic document
- Investigating errors
- Terminating the Toolkit

```
main()
{
dla_toolkit_handle          toolkit_hdl;
dla_document_handle         document_hdl;
dla_entity_handle           doc_prof_hdl;
dla_dap_level               dap_level;
char                        *input_file = "infile.odf";
dla_length                  ident_buffer_length = 50;
dla_octet_string            ident_return_buffer  [50];
dla_length                  ident_length_return;
dla_entity_type             entity_type_return;
dla_property_code           property_code_return;
dla_integer                 dla_err_num_return;
ola_constituent_type        constituent_type_return;
ola_integer                 byte_position_return;
ola_attribute_code          attribute_code_return;
ola_integer                 ola_err_num_return;

return_status =
    dla_init_toolkit(
          DLA_C_NO_OPTIONS,
          0,
          0,
          &toolkit_hdl);

return_status =
    dla_read_document(
          toolkit_hdl,
          input_file,
          strlen(input_file),
          DLA_C_NO_OPTIONS,
          DLA_C_NULL_STRING,
          0,
          &dap_level,
          &document_hdl);

if (dla_failed(return_status) &&
    dla_error_class(return_status) == DLA_C_INTERP_ERROR )
{
    return_status =
          dla_get_interp_error(
                toolkit_hdl,
                ident_buffer_length,
                ident_return_buffer,
                &ident_length_return,
                &entity_type_return,
                &property_code_return,
                &dla_err_num_return);

          /* code to display or capture reason for failure */
```

```
}
else
if (dla_failed(return_status) &&
    dla_error_number(return_status) ==
          DLA_E_INV_ODIF)
{
    return_status =
          dla_get_ola_read_error(
                toolkit_hdl,
                &constituent_type_return,
                &byte_position_return,
                &attribute_code_return,
                &ola_err_num_return);

          /* code to display or capture the reason for failure */
}

return_status =
    dla_find_profile(
          document_hdl,
          &doc_prof_hdl);
/*
 * The application now determines, by examining properties in
 * the document profile, whether an external document or
 * resource document is referenced in the document, and what
 * the file name of this generic document is. The generic
 * document can then be read as the corresponding type,
 * which is external in this example.
 */
return_status =
    dla_read_generic_doc(
          document_hdl,
          "extdoc",
          strlen("extdoc"),
          DLA_C_EXTERNAL_DOC,
          &ext_doc_hdl);

/*
 * Now process the document...
 *
 * Terminate the toolkit.
 */

return_status =
    dla_term_toolkit(toolkit_hdl);
}
```

### 2.2.11 Navigating Logical Structures

This subclause provides the following two examples:

- Traversing an arbitrary specific logical structure
- Locating a parent entity

Navigating the specific layout structure is done in exactly the same way as navigating the specific logical structure.

#### 2.2.11.1 Structure Traversal

The following code fragments illustrate how to traverse a specific logical structure, for instance the structure in 2.2.12.

```
walk_structure(
    dla_document_handle         document_hdl,
    dla_entity_handle           start_node)
{
    dla_entity_type             node_type;
    dla_entity_handle           next_node;
    dla_entity_handle           next_sibling_node;

/* Determine the type of start_node */

    return_status =
          dla_get_entity_type(
                document_hdl,
                start_node,
                &node_type);

    switch (node_type)
    {
    case DLA_C_SPEC_LOG_ROOT:
    case DLA_C_SPEC_PASSAGE:
    case DLA_C_SPEC_NUM_SEG:
    case DLA_C_SPEC_NUMBER:
    case DLA_C_SPEC_PARAGRAPH:
    case DLA_C_SPEC_FOOTNOTE:
    case DLA_C_SPEC_FOOTNOTE_REF:
    case DLA_C_SPEC_FOOTNOTE_NUM:
    case DLA_C_SPEC_FOOTNOTE_BODY:
    case DLA_C_SPEC_FOOTNOTE_TEXT:
    case DLA_C_SPEC_BODY_TEXT:
    case DLA_C_SPEC_BODY_GEOM:
    case DLA_C_SPEC_BODY_RASTER:

/* Do any processing specific to the type of entity */

    default:
          break;
    }

    /* Traverse the subordinates of start_node */

    return_status =
          dla_move_to_child_spec_entity(
                document_hdl,
                start_node,
                DLA_C_CHILD_FIRST,
                &next_node);

    if (  dla_failed(return_status) &&
          (dla_error_number (return_status) ==
          DLA_E_ENTITY_HAS_NO_CHILD) )
    {
    /*
```

```
        * No children - do any processing required for this
        * special case.
        */
       }
       else
       {
              while (TRUE)
              {
                     walk_structure(document_hdl, next_node);
                     return_status =
                            dla_move_to_sibling_spec_entity(
                                   document_hdl,
                                   next_node,
                                   DLA_C_SIB_NEXT,
                                   &next_sibling_node);

                     if ( (dla_failed(return_status)
                            && (dla_error_number(return_status) ==
                                   DLA_E_ENTITY_ONLY_CHILD) )
                          ||
                          (!(dla_failed(return_status))
                            && (next_sibling_node == DLA_C_NULL_HANDLE) )
                        )
                     {
                     /* There are no more siblings */
                            break;
                     }
                     next_node = next_sibling_node;
              }
       }
}

/* First move to the DocumentLogicalRoot */

return_status =
    dla_move_to_root_entity(
           document_hdl,
           DLA_C_SPECIFIC_LOGICAL,
           &doclogroot_hdl);

walk_structure(document_hdl, doclogroot_hdl);
```

#### 2.2.11.2 Location of ParentEntity

This subclause shows how to determine the handle of the parent entity of a given entity.

```
dla_entity_handle              current_entity_hdl;
dla_entity_handle              parent_entity_hdl;

return_status =
    dla_move_to_parent_spec_entity(
           document_hdl,
           current_entity_hdl,
           &parent_entity_hdl);
```

### 2.2.12 Navigating Generic Structures

This subclause contains examples to show the following:

- Location by identifier
- Location by sequence
- Downward navigation of the generic layout structure
- Upward navigation of the generic layout structure

Navigation of the generic logical structure is performed in the same way as navigation of the generic layout structure.

#### 2.2.12.1 Location by Identifier

The subclause demonstrates the location by identifier of an entity in the generic layout structure.

```
dla_entity_handle               entity_handle;
dla_entity_type                 entity_type;

return_status =
    dla_locate_constit(
          document_hdl,
          "0 1",
          strlen("0 1"),
          &entity_handle,
          &entity_type);
```

If there is an entity in the document that is referenced by the document_hdl argument with the identifier "0 1", a handle that references that entity is returned in the entity_handle argument. Its entity type is returned in the argument entity_type.

#### 2.2.12.2 Location by Sequence

The functions dla_find_first_entity and dla_find_next_entity can be used to find all generic layout or generic logical entities in a document.

```
dla_status                      return_status;
dla_document_handle             document_hdl;
dla_entity_handle               gen_log_ent_hdl;

/* Find the first generic logical entity */

return_status =
    dla_find_first_entity(
          document_hdl,
          DLA_C_GENERIC_LOGICAL,
          &gen_log_ent_hdl);

/*
 * Find the rest of the generic logical entities
 */
while (gen_log_ent_hdl != DLA_C_NULL_HANDLE)
{
    return_status =
          dla_find_next_entity(
                document_hdl,
                gen_log_ent_hdl,
                &gen_log_ent_hdl);
}
```

#### 2.2.12.3 Downward Navigation

This example demonstrates the determination of the first frame in the Footer area of a VersoPage in the first Pageset of a document.

It consists of the following steps:

- Determining the handle of the generic DocumentLayoutRoot
- Determining the handle of the first Pageset
- Determining the generic Page entities that make up the Pageset
- Determining the handle of the Footer area, if it exists
- Determining the type of the Footer area, assuming that it is a BasicFooter
- Determining the number of entities associated with the BasicFooter frame

```
dla_entity_handle             genlayroot_hdl;
dla_property_status           property_status;
dla_derived_location          derived_location;
dla_entity_handle             derived_hdl;
dla_integer                   sequence_size;
dla_entity_handle             pageset_hdl;
dla_integer                   choice_id;
dla_entity_handle             page_hdl;
dla_entity_handle             footer_hdl;
dla_entity_type               entity_type;

/*
 * Determine the handle of Generic DocumentLayoutRoot.
 * Assume that document_hdl has been returned by
 * dla_read_document.
 */

return_status =
    dla_move_to_root_entity(
          document_hdl,
          DLA_C_GENERIC_LAYOUT,
          &genlayroot_hdl);

/* Determine the number of PageSets. */

return_status =
    dla_get_spec_size(
          document_hdl,
          genlayroot_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_PGE_SET_LST,
          &sequence_size,
          &property_status);

/*
 * Assume that there is at least one Pageset.
 * Get the handle of the first PageSet
 */

return_status =
    dla_get_spec_index_entity(
          document_hdl,
          genlayroot_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_PGE_SET,
          1,
          &pageset_hdl,
          &property_status);

/*
 * Determine whether the Pageset contains Recto and Verso
 * pages.
 */

return_status =
    dla_get_spec_choice(
          document_hdl,
          pageset_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_GEN_REM_PGE,
          choice_id,
          &property_status);

/* Assume that choice_id is DLA_C_PGE_REC_VER */

/* Get the handle of the VersoPage */
```

```
return_status =
    dla_get_spec_entity_handle(
          document_hdl,
          pageset_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_GEN_VER_PGE,
          &page_hdl,
          &property_status);

/* Get the handle of the footer area */

return_status =
    dla_get_spec_entity_handle(
          document_hdl,
          page_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_PGE_FOT,
          &footer_hdl,
          &property_status);

/* Determine the type of the footer area */

return_status =
    dla_get_entity_type(
          document_hdl,
          footer_hdl,
          &entity_type);

/*
 * Assume that entity type is DLA_C_GEN_BASIC_FOOTER.
 * There is therefore only one frame.
 */
```

#### 2.2.12.4 Upward Navigation

This example illustrates the determination of the parent VariableCompositeBody frames of a SnakingColumn frame. It consists of the following steps:

- Getting the number of VariableCompositeBody frames that are parent to the SnakingColumn frame
- Getting the handle of each VariableCompositeBody frame

```
/* Get the number of parent frames */

return_status =
    dla_get_spec_size(
          document_hdl,
          snaking_col_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_PAR_ENT_LST,
          &sequence_size,
          &property_status);
/* Retrieve each parent entity handle */
for(i=1; i<=sequence_size; i++)
{
    return_status =
          dla_get_spec_index_entity(
                document_hdl,
                snaking_col_hdl,
                DLA_C_NULL_HANDLE,
                DLA_C_PAR_ENT,
                i,
                &parent_entity_hdl,
                &property_status);

    /* Process each parent entity as required */
}
```

### 2.2.13 Property Getting

This subclause illustrates the getting of properties of DAP entities.

It demonstrates the property getting of the following properties:

- User visible name on a Numbered Segment, with and without defaulting
- Initial value for a Segment Number
- All the Tabulation stops for an entity
- The graphic character sets designated for an entity
- The value of the layout directive property New layout object

#### 2.2.13.1 Get User Visible Name with Defaulting

This example illustrates getting the value of the property User visible name on a Numbered Segment with defaulting. It consists of the following steps:

- Getting the length of the string
- Getting the value of the string

```
    /* Get the length of the string */

dla_octet_string            string_name;
dla_length                  string_length;
dla_length                  actual_length;
dla_derived_location        derived_location;
dla_entity_handle           derived_entity_handle;

return_status =
    dla_get_length(
          document_hdl,
          numseg1_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_USR_NAM,
          &string_length,
          &property_status,
          &derived_location,
          &derived_entity_hdl);

    /* Allocate string_length octets pointed to by string_name */

    /* Get the value of the string */

return_status =
    dla_get_string(
          document_hdl,
          numseg1_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_USR_NAM,
          string_length,
          string_name,
          &actual_length,
          &property_status,
          &derived_location,
          &derived_entity_hdl);

/*
 * If the derived_location is ODA standard (DLA_C_DEF_STD_DEF) then
 * the string should be "empty string".
 */
```

#### 2.2.13.2 Get User Visible Name without Defaulting

This example illustrates getting the value of the property User visible name on a Numbered Segment without defaulting. It consists of the following steps:

- Getting the length of the string
- Getting the value of the string

```
/* Get the length of the string */

dla_octet_string                string_name;
dla_length                      string_length;
dla_length                      actual_length;

return_status =
    dla_get_spec_length(
          document_hdl,
          numseg1_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_USR_NAM,
          &string_length,
          &property_status);

/* Allocate string_length octets pointed to by string_name */

/* Get the value of the string */

return_status =
    dla_get_spec_string(
          document_hdl,
          numseg1_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_USR_NAM,
          string_length,
          string_name,
          &actual_length,
          &property_status);
```

#### 2.2.13.3 Get Reset Value for Segment Number

This example illustrates how to get the Reset value for a Segment Number.

The Numbered Segment whose Reset value is required is numseg21, which is included in the example in 2.2.6.1.

The example consists of the following steps:

- Getting the number of structures for Segment Number reset on the entity numseg2
- For each structure:
  - Getting the level to which the structure applies
  - Getting the Reset value for level 2

It is assumed that values on the classes (initialization) have already been checked, using similar code. This scan need only be done once for each level in a passage.

```
dla_property_status             property_status;
dla_derived_location            derived_location;
dla_entity_handle               derived_entity_hld;
dla_integer                     structure_size;
dla_structure_handle            structure_hdl;
dla_integer                     integer_var;
dla_integer                     init_val_on_obj;
dla_integer                     init_val_on_class;

/* Get the number of structures */

return_status =
    dla_get_spec_size(
          document_hdl,
          numseg2_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_SEG_NUM_RESET_LST,
          &structure_size,
          &property_status);
```

```
for (i=1; i <= structure_size; i++)
{
    /* Get the structure handle */

    return_status =
          dla_get_spec_index_structure(
                document_hdl,
                numseg2_hdl,
                DLA_C_NULL_HANDLE,
                DLA_C_SEG_NUM_INIT,
                &structure_hdl,
                i,
                &property_status);

    /* Initialization (reset) on the object */

    /* Get the level number */

    return_status =
          dla_get_spec_integer(
                document_hdl,
                num_seg2_hdl,
                structure_hdl,
                DLA_C_SEG_NUM_LEV_NUM,
                &integer_var,
                &property_status);

    if (integer_var == 2)
    { /* correct level of Numbered Segment */

          /* Get the Reset Value */

    return_status =
          dla_get_spec_integer(
                document_hdl,
                num_seg2_hdl,
                structure_hdl,
                DLA_C_SEG_NUM_RESET_VAL,
                &init_val_on_obj,
                &property_status);

          /*
           * If this call worked, then we have found the
           * reset value so break.
           */
    }
}
```

#### 2.2.13.4 Get Tabulation Stops

This example illustrates how to get, without defaulting, all the tabulation stops for a specific logical entity, text2. It consists of the following steps:

- Getting the number of tabulation stops
- Getting each tabulation stop and its associated strings

```
dla_length                  tab_reference_length;
dla_length                  tab_reference_length_return;
dla_integer                 tab_position;
dla_tab_alignment           tab_alignment;
dla_length                  tab_alignment_length;
dla_length                  tab_alignment_length_return;
dla_octet_string            tab_reference;
dla_octet_string            tab_alignment_string;
```

```
/* Get the number of tabulation stops */

return_status =
    dla_get_spec_size(
          document_hdl,
          text2_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_CA_LIN_LAY,
          &sequence_size,
          &property_status);

for(i=1; i<=sequence_size; i++)
{
    return_status =
          dla_get_spec_tab_stop(
                document_hdl,
                text2_hdl,
                i,
                &tab_reference_length,
                &tab_position,
                &tab_alignment,
                &tab_alignment_length);

    /*
     * Allocate tab_reference_length octets pointed to by
     * tab_reference and tab_alignment octets pointed to by
     * tab_alignment_string.
     */

    return_status =
          dla_get_spec_tab_stop_strings(
                document_hdl,
                text2_hdl,
                i,
                tab_reference_length,
                tab_alignment_length,
                tab_reference,
                &tab_reference_length_return,
                tab_alignment_string,
                &tab_alignment_length_return);
}
```

### 2.2.13.5 Get Graphic Character Sets

This example illustrates how to get the Graphic character sets designated for an entity text2, and also the Code extension announcers for text2.

```
#DEFINE MAX_SEQ_COUNT  5        /* Maximum number of
                                escape sequences */

dla_octet_string                char_set_id[MAX_SEQ_COUNT];
dla_integer                     code_area[MAX_SEQ_COUNT];
dla_integer                     char_set_type[MAX_SEQ_COUNT];
dla_integer                     counter;
dla_octet_string                code_ext_ann;
dla_length                      code_ext_ann_length;
dla_length                      code_ext_ann_length_return;
dla_length                      char_set_id_length;
dla_length                      char_set_id_buflen = 1;

/*
 * The last character identifier is one character so ensure
 * that each element of char_set_id points to a buffer of at
 * least one character.
 */
```

```
return_status =
    dla_get_char_set_count(
           document_hdl,
           entity_hdl,
           DLA_C_CA_GR_CHR_SET,
           &counter,
           &property_status,
           &derived_location,
           &derived_entity_hdl);

for (i = 0; (i < MAX_SEQ_COUNT)  && (i < counter); i++)
{
    return_status =
           dla_get_nth_char_set(
                  document_hdl,
                  entity_hdl,
                  DLA_C_CA_GR_CHR_SET,
                  i + 1,
                  char_set_id_buflen,
                  char_set_id[i],
                  &char_set_id_length,
                  &code_area[i],
                  &char_set_type_[i],
                  &property_status,
                  &derived_location,
                  &derived_entity_hdl);
}

/* Now get the code extension announcers. */

return_status =
    dla_get_length(
           document_hdl,
           text2,
           DLA_C_NULL_HANDLE,
           DLA_C_CA_COD_EXT_ANN,
           &code_ext_ann_length,
           &property_status,
           &derived_location,
           &derived_entity_hdl);

/*
 * Allocate code_ext_ann_length octets pointed to by
 * code_ext_ann.
 */

/* Get the value of the string */

return_status =
    dla_get_string(
           document_hdl,
           text2_hdl,
           DLA_C_NULL_HANDLE,
           DLA_C_CA_COD_EXT_ANN,
           cod_ext_ann_length,
           cod_ext_ann,
           &cod_ext_ann_length_return,
           &property_status,
           &derived_location,
           &derived_entity_hdl);
```

#### 2.2.13.6 Get New Layout Object

This example illustrates getting the value, without defaulting, of the layout directive property New layout object, for a specific logical entity, numseg1.

It consists of the following steps:

- Checking that the property is present
- Getting the choice of property
- Retrieving the property value for each possible choice

```
dla_integer                    choice_identifier;

    /* Check that the property is present */

return_status =
    dla_get_spec_status(
          document_hdl,
          numseg1_hdl
          DLA_C_NULL_HANDLE,
          DLA_C_LYD_NEW_LAY_OBJ,
          &property_status);

if (property_status != DLA_C_PROP_SPECIFIED)
{
    /* Call exception handling code */
}

/* Determine which CHOICE is present */

return_status =
    dla_get_spec_choice(
          document_hdl,
          numseg1_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_LYD_NEW_LAY_OBJ,
          &choice_identifier,
          &property_status);

/*
 * For each possible CHOICE, call the function to get the
 * property.
 */

switch (choice_identifier)
{
case DLA_C_NEW_PGE_SET:
    return_status =
          dla_get_spec_entity_handle(
                document_hdl,
                numseg1_hdl,
                DLA_C_NULL_HANDLE,
                DLA_C_LYD_NEW_PGE_SET,
                &entity_hdl,
                &property_status);

    /* Add property specific code here */
    break;

case DLA_C_PGE_BRK:

    /* No code required to get value of property */
    /* Add property specific code here */
    break;
```

```
case DLA_C_NEW_PGE:
    return_status =
          dla_get_spec_entity_handle(
                document_hdl,
                numseg1_hdl,
                DLA_C_NULL_HANDLE,
                DLA_C_LYD_NEW_PGE,
                &entity_hdl,
                &property_status);

    /* Add property specific code here */
    break;

case DLA_C_COL_BRK:
    return_status =
          dla_get_spec_entity_handle(
                document_hdl,
                numseg1_hdl,
                DLA_C_NULL_HANDLE,
                DLA_C_LYD_COL_BRK,
                &entity_hdl,
                &property_status);

    /* Add property specific code here */
    break;

case DLA_C_NEW_LAY:
    return_status =
          dla_get_spec_entity_handle(
                document_hdl,
                numseg1_hdl,
                DLA_C_NULL_HANDLE,
                DLA_C_LYD_NEW_LAY,
                &entity_hdl,
                &property_status);

    /* Add property specific code here */
    break;

default:
    /* Error handling code */
    break;
}
```

### 2.2.14 Reading Content

This subclause illustrates the following:

- Reading content for a FootnoteBody
- Reading content for a Header
- Converting content for a Paragraph

#### 2.2.14.1 Reading Content for a FootnoteBody

This example reads the content associated with a footnote in a FOD26 document.

```
dla_entity_handle           fnote_body_hdl;
dla_entity_handle           fnote_sub_hdl;
dla_entity_handle           fnote_text_hdl;

/*
 * Get the first subordinate of the FootnoteBody; i.e. the
 * FootnoteReference and then first FootnoteText following it.
 *
 * Assume that the handle of the FootnoteBody is fnote_body_hdl
 */
```

```
return_status =
    dla_get_spec_entity_handle(
          document_hdl,
          fnote_body_hdl,
          DLA_C_NULL_HANDLE,
          DLA_C_ENT_FNOTE_NUM,
          &fnote_sub_hdl,
          &property_status);

while (fnote_sub_hdl != DLA_C_NULL_HANDLE)
{
    dla_octet_string                buffer_ptr;
    dla_length                      buffer_len;
    dla_length                      content_len;

    /* Get the next basic logical entity */

    return_status =
          dla_move_to_sibling_spec_entity(
                document_hdl,
                fnote_sub_hdl,
                DLA_C_SIB_NEXT,
                &fnote_sub_hdl);

    fnote_text_hdl = fnote_sub_hdl;

    /* Get the length of the content */

    return_status =
          dla_get_content_length(
                document_hdl,
                fnote_text_hdl,
                &content_len);

    /* Allocate a buffer of the required length */

    /* Read the content */

    return_status =
          dla_get_content(
                document_hdl,
                fnote_text_hdl,
                0,
                buffer_len,
                buffer_ptr,
                &content_len);

    /* Process the content */

}
```

#### 2.2.14.2 Reading Content for a Header

The following code fragment reads the content associated with a generic BasicFooter whose handle is footer_hdl. The content could be in a resource document.

```
/*
 * Find the number of Common generic logical entities
 * associated with footer area.
 */
```

```
return_status =
    dla_get_spec_size(
           document_hdl,
           footer_hdl,
           DLA_C_NULL_HANDLE,
           DLA_C_COM_CON_ENT_LST,
           &sequence_size,
           &property_status);

for (i=1; i<=sequence_size; i++)
{
    dla_octet_string                buffer_ptr;
    dla_length                      buffer_len;
    dla_length                      content_len;

    return_status =
           dla_get_spec_index_entity(
                  document_hdl,
                  footer_hdl,
                  DLA_C_NULL_HANDLE,
                  DLA_C_COM_CON_ENT,
                  i,
                  &entity_hdl,
                  &property_status);

    /* Get entity_type of entity */

    return_status =
           dla_get_entity_type(
                  document_hdl,
                  entity_hdl,
                  &entity_type);

    /* Get the length of the content */

    if (entity_type != DLA_C_GEN_PAGE_NUMBER)
    {

           return_status =
                  dla_get_content_length(
                         document_hdl,
                         entity_hdl,
                         &buffer_len);

           /* Allocate a buffer of the required length */

           /* Read the content */

           return_status =
                  dla_get_content(
                         document_hdl,
                         entity_hdl,
                         0,
                         buffer_len,
                         buffer_ptr,
                         &content_len);
    }
}
```

#### 2.2.14.3 Converting Content for a Paragraph

This example converts the content associated with a Paragraph in a document at DAP level 2. Character content is converted from ISO 6937/2 to ISO 8859-1 and Raster content is converted to bitmap-encoding.

```
/*
 * Get the first subordinate of the Paragraph
 */
return_status =
    dla_move_to_child_spec_entity(
          document_hdl,
          para_hdl,
          DLA_C_CHILD_FIRST,
          &entity_hdl);

while (entity_hdl != DLA_C_NULL_HANDLE)
{
    /* get the entity type */

    return_status =
          dla_get_entity_type(
                document_hdl,
                entity_hdl,
                &entity_type);

    switch (entity_type)
    {
    case DLA_C_SPEC_BODY_TEXT:
          return_status =
                dla_convert_content(
                      document_hdl,
                      entity_hdl,
                      DLA_C_CHAR_6937_TO_8859);
          break;
    case DLA_C_SPEC_BODY_RASTER:
          return_status =
                dla_convert_content(
                      document_hdl,
                      entity_hdl,
                      DLA_C_RAS_TO_BIT);
          break;
     case DLA_C_FOOTNOTE:
          /*
           * Depending on whether it is required to convert the
           * content for the FootnoteBody, navigate to the
           * subordinate FootnoteText entities.
           */
          break;
     default:
          break;
     }

     /* Get the next basic logical entity */

     return_status =
          dla_move_to_sibling_spec_entity(
                document_handle,
                entity_hdl,
                DLA_C_SIB_NEXT,
                &entity_hdl);
  }
```

## 2.3 Restrictions

This subclause gives information on the following areas of restriction:

- Architectural, or data stream restrictions on documents
- API, or operational restrictions on document manipulation

### 2.3.1 Architectural Restrictions on Documents

Any generic document used with the DAP API as a resource or external document must identify itself in the document profile as being of the same DAP as the specific document.

The FODs allow a form of processable document that has no generic layout structure. Such a document can have only very limited layout directives and is not formattable. It is not like a document with an external document class, in that no references to layout constituents (or in DAP API terms, entities) are permitted, so the missing generic layout structure cannot be supplied by simple addition. FOD36 has mandatory layout directives on some constituent constraints that make a generic layout structure effectively mandatory too. Such incomplete documents cannot be read or created with the DAP API, except as resource documents.

The common content of headers and footers in level 2 and 3 DAPs can be carried in either processable or formatted (*arranged* in FOD terminology) varieties. Only processable common content can be created, but both varieties can be read.

FOD26 allows the use of ORDINAL in segment numbering binding expressions, and both FOD11 and FOD26 allow its use in page numbering binding expressions. Such usage does not, in this context, provide any functionality that is not possible with incrementation of the corresponding binding value. For documents conforming to FOD11 or FOD26, the DAP API does not create or expose such a usage. This does not lead to misinterpretation of such documents, except in the following exceptional case: if a document contains an initialization or resetting of a numbering binding that is redundant due to the use of ORDINAL, the DAP API presents this initialization or resetting as if it were effective. For FOD36 documents, the use of ORDINAL is exposed or may be requested through the ORD expression type of the general numbering properties (see A.2.5.3).

In FOD36 it is permitted to have more than one subordinate footnote frame within a single body frame or composite column frame. The implications of such a structure for the layout process are complex, and the usefulness is limited. It is expected that where more than one footnote frame is required on a page, each such generic frame will belong to its own generic column area, for instance for polyglot multiple columns. Thus only a single generic footnote frame can be linked subordinate to any other generic frame, although documents with multiple footnote frame subordinates can be read.

The ODA attribute Synchronization is unrestricted in form by FOD26 and FOD36. See Appendices C & D for the FOD clarifications that have been adopted. Only the form with a simple entity reference is handled either for reading or writing and so the handling of Synchronization is limited. The alternative form is an object expression that would require full expression evaluation when reading and either full application responsibility or a set of predefined expressions when writing.

When writing documents in mode-1, the limitations to the creation of documents that are implied by the automatic handling and hiding of generic logical structure and styles are explained in 2.1. They do not limit the way in which properties can be applied to document parts. The limitation is only to the control of the different ways in which the properties can be specified for ODA constituents, which gives rise to their application to entities.

Where several frames have or are to have the same category, this can be detected by the use of distinct properties or established by the use of the function dla_make_frame_equiv respectively. The condition where any frame has more than one permitted category cannot be established when writing documents nor detected when reading documents.

Entities of type Reference form a cluster with their single subordinate entity of type ReferencedContent. The DAP API does not provide for the creation of other basic logical entities as subordinates to Reference. The functionality of these items (prefix and suffix text) is provided for by the use of bindings containing character strings that are concatenated with the generated content.

In FOD36, the DAP type Number appears in NumberedSegment, Figure and NumberedList. There is also an annotation in FOD36 for the macro SegmentNumbers, which says that B_REF(CURR_OBJ) is used for list numbers. As a consequence of this usage, a single generic Number entity in the generic logical structure cannot be in the Generator for subordinates of both a generic Figure and a generic NumberedList.

No use is made of unit-scaling for FOD36. It is not clear how the font metrics and permissible and non-basic values are to be interpreted in the presence of this attribute. No corresponding property is supported.

The possible subordinates of each Form and each EntryGroup consist of an arbitrary set (ODA construction type AGG) of entities of types EntryGroup and EntryElement. As the term AGG means that instances must consist of exactly one of each of the constituents whose classes are in the AGG expression, the DAP API prevents the creation of documents where this is not the case. Although the order of the instances is nominally free, the DAP API insists on the subordinates of instances of any given class all being in the same order. This should be of no real significance to the application, as it is the position of the FormEntryArea frames that determines the eventual appearance of the Form.

### 2.3.2 API Restrictions on Document Manipulation

The DAP API does not attempt to ensure that a created document is formattable without errors.

The writing of the specific layout structure is not supported by the DAP API, so it cannot be used to create formatted processable and formatted form documents.

When reading a document, the DAP API generally requires and expects that the document conforms to a supported DAP as indicated in its profile. If this is not the case, some reading operations may fail if the document cannot be interpreted. The DAP API does not undertake to perform full conformance checking or to provide any indication of the non-conformance, other than a return code from the failing operation. Constraints that are implied but not stated in the FODs (such as that if a generic layout structure is present then it must include within its Permitted categories attributes all layout categories that appear in layout styles) must also be met.

When reading formatted form documents, it can be assumed that any factorization is solely for the purpose of economy of transmission and possibly of storage. The ability to read Position and Dimension properties of generic layout entities of formatted form documents is not needed Therefore, the DAP API does not provide access to the property-entity combinations that would be required to read factorized properties on the generic entities for which they may be specified. It is equally efficient for applications to read the properties from the specific layout entities using the defaulting functions. The few properties that are conformant for both formatted processable and formatted form documents can be read from the generic layout entities in the usual way.

When writing a document, the number of layout categories generated when creating a generic layout structure is limited to 99 999 999. This places a limit on the total number of lowest level frames, discounting the second and subsequent column variable frames belonging to a single snaking frame, which use only one category between them.

The DAP API does not provide a function to delete or to unlink DAP entities.

The DAP API does not check any aspects of content information for conformance.

When a pre-existing document is read, it is placed in a read-only state. Changes cannot be made to a document when it is in this state. It is not therefore possible to use the DAP API to edit a document, except by using it to construct a new document modelled on the original, and then editing this copy.

When reading a document that refers to a resource document and setting up the numbering initialization properties, the DAP API only consults bindings in resource generic documents for resource entities belonging to those generic entities that have no bindings in the interchanged document.

When writing a document that refers to an external document class, any styles in that external document class can only be used through references within the generic logical and layout structures. Individual styles cannot be referred to directly.

In an input ODA document, the combination of the constraint name parameter of the attribute Application comments and the constituent type is interpreted by the DAP API to identify the entity type of each entity. This is an essential process before other aspects of each entity can be interpreted. The derivation of this information may require the presence of one or both generic structures. For this reason, if a document requires an external generic document then all access to the specific structures is prevented until the generic document has been read.

When adding tab stops during the writing of a document, the DAP API Tool adds the tab stop to the invisible line layout table that applies to that entity. Tab stops cannot be removed from a line layout table or modified within it.

There is no default layout path property in the document profile. This is because of the absence of a context in which such a property could be interpreted in line with the frame property Layout direction (DLA_C_LAY_DIR). The default value may be present in externally sourced documents and is included in the defaulting (by the ODA API).

There are a number of places within Table and its subordinates where the Generator for subordinates is of the form REP or REPCHO. Although this implies that there must be at least one subordinate from the construction-term, the DAP API does not automatically create a subordinate. That is, composite entities that potentially have Generator for subordinates expressions of this form are not regarded as clumps. In this respect their treatment is the same as if the expression had been OPTREP, but the DAP API creates the correct conformant expression.

FOD36 permits the use of Default value lists (DVL) on CommonContent. The defaulting process for DVL requires a specific structure in order to define the particular superior/subordinate relationship upon which the DVL mechanism depends. As the generic structure is not guaranteed to be a tree, the parent of any of the basic classes cannot be unambiguously defined, and hence the applying DVL cannot be determined. DVL operation depends on specific structure, which for CommonContent only exists ephemerally during the layout process.

The use of the DVL attribute for CommonContent components is deprecated although the establishment and reading of the corresponding two properties for these entities is not prevented by the DAP API. Property values in styles specified by CommonContent DVL are not (or cannot) be returned by reading the property code with defaulting.

# 3 DAP Level API Reference

This clause provides a description of the application programming interfaces. It is divided into the following sections:

- Types
- Constants
- Error Handling
- Functional Specifications

## 3.1 Types

This subclause provides details for the following data types:

- Basic data types
- Enumerated data types

### 3.1.1 Basic Data Types

Table 3-1 lists the basic data types.

**Table 3-1 Basic Data Types**

| Basic data type | Meaning |
|---|---|
| dla_entity_handle | The handle of a constituent of a DAP-conformant ODA document.<br>DLA_C_NULL_HANDLE indicates a null handle. |
| dla_document_handle | The handle of a DAP-conformant ODA document. |
| dla_integer | A 32-bit signed integer. |
| dla_length | A 32-bit unsigned integer. |
| dla_octet_string | A string of octets. DLA_C_NULL_STRING indicates a null string. |
| dla_toolkit_handle | The handle of a toolkit instance. |
| dla_null | Null-valued type. |
| dla_structure_handle | The handle of a structure. DLA_C_NULL_HANDLE indicates a null handle. |
| dla_callback_function | A pointer to a callback function or DLA_C_NULL_FUNCTION. DLA_C_NULL_FUNCTION indicates that no callback is required. |

### 3.1.2 Enumerated Data Types

Table 3-2 lists the enumerated data types.

**Table 3-2 Enumerated Data Types**

| Enumerated data type | Meaning |
|---|---|
| dla_dap_level | DAP level |
| | DLA_C_DAP_LEVEL_1<br>DLA_C_DAP_LEVEL_2<br>DLA_C_DAP_LEVEL_3 |
| dla_entity_type | Type of entity |
| | DLA_C_TYP_UNDEFINED (error) |
| | DLA_C_PROFILE<br>DLA_C_TYP_LAY_STYLE<br>DLA_C_TYP_PRES_STYLE |
| | DLA_C_SPEC_LOG_ROOT<br>DLA_C_SPEC_PASSAGE<br>DLA_C_SPEC_NUM_SEG<br>DLA_C_SPEC_NUMBER<br>DLA_C_SPEC_TITLE<br>DLA_C_SPEC_CAPTION<br>DLA_C_SPEC_PARAGRAPH<br>DLA_C_SPEC_PHRASE<br>DLA_C_SPEC_FOOTNOTE<br>DLA_C_SPEC_FOOTNOTE_REF<br>DLA_C_SPEC_FOOTNOTE_NUM<br>DLA_C_SPEC_FOOTNOTE_BODY<br>DLA_C_SPEC_FOOTNOTE_TEXT<br>DLA_C_SPEC_FIGURE<br>DLA_C_SPEC_BODY_TEXT<br>DLA_C_SPEC_REFERENCE<br>DLA_C_SPEC_REF_CONTENT<br>DLA_C_SPEC_BODY_RASTER<br>DLA_C_SPEC_BODY_GEOM<br>DLA_C_SPEC_DESCRIPTION<br>DLA_C_SPEC_ARTWORK<br>DLA_C_SPEC_NUMBERED_LIST<br>DLA_C_SPEC_UNNUMBERED_LIST<br>DLA_C_SPEC_DEFINITION_LIST<br>DLA_C_SPEC_LIST_ITEM<br>DLA_C_SPEC_LIST_TERM<br>DLA_C_SPEC_TABLE<br>DLA_C_SPEC_ROW<br>DLA_C_SPEC_TABLE_COMPONENT<br>DLA_C_SPEC_ROW_COMPONENT<br>DLA_C_SPEC_FORM<br>DLA_C_SPEC_ENTRY_ELEMENT<br>DLA_C_SPEC_ENTRY_GROUP<br>DLA_C_SPEC_ENTRY_TEXT<br>DLA_C_SPEC_ENTRY_RASTER<br>DLA_C_SPEC_ENTRY_GEOMETRIC |

Table 3-2 (cont.) Enumerated Data Types

| Enumerated data type | Meaning |
|---|---|
| dla_entity_type (cont.) | DLA_C_GEN_LAY_ROOT<br>DLA_C_GEN_PAGESET<br>DLA_C_GEN_PAGE<br>DLA_C_GEN_RECTO_PAGE<br>DLA_C_GEN_VERSO_PAGE<br>DLA_C_GEN_COMP_HEADER<br>DLA_C_GEN_COMP_BODY_FIX<br>DLA_C_GEN_COMP_BODY_VAR<br>DLA_C_GEN_COL_FIXED<br>DLA_C_GEN_COL_VARIABLE<br>DLA_C_GEN_SNAKING_COLUMN<br>DLA_C_GEN_SYNC_COLUMN<br>DLA_C_GEN_BASIC_FLOAT<br>DLA_C_GEN_COMP_FLOAT<br>DLA_C_GEN_BASIC_COLUMN<br>DLA_C_GEN_FOOTNOTE_AREA<br>DLA_C_GEN_ARRNGD_CONT_FIXED<br>DLA_C_GEN_ARRNGD_CONT_VAR<br>DLA_C_GEN_SRCD_CONT_FIXED<br>DLA_C_GEN_SRCD_CONT_VAR<br>DLA_C_GEN_COMP_FIXTURE_VAR<br>DLA_C_GEN_COMP_FIXTURE_FIXED<br>DLA_C_GEN_BASIC_FIXTURE<br>DLA_C_GEN_COMP_COL_FIXED<br>DLA_C_GEN_COMP_COL_VARIABLE<br>DLA_C_GEN_COMP_COMMON<br>DLA_C_GEN_COMP_ARTWORK<br>DLA_C_GEN_BASIC_HEADER<br>DLA_C_GEN_BASIC_BODY<br>DLA_C_GEN_GENERIC_BLOCK<br>DLA_C_GEN_FORM_AREA<br>DLA_C_GEN_COMP_FOOTER<br>DLA_C_GEN_BASIC_FOOTER<br>DLA_C_GEN_TABLE_HEADER<br>DLA_C_GEN_ENTRY_GROUP_AREA<br>DLA_C_GEN_TABLE_AREA<br>DLA_C_GEN_TABLE_LABEL<br>DLA_C_GEN_COMP_TABLE_LABEL<br>DLA_C_GEN_LABEL_COMPONENT<br>DLA_C_GEN_ROW_AREA<br>DLA_C_GEN_CELL<br>DLA_C_GEN_SUBROW_GROUP<br>DLA_C_GEN_SUBROW<br>DLA_C_GEN_TABLE_LABEL_CONTENT<br>DLA_C_GEN_FORM_ENTRY_AREA<br><br>DLA_C_SPEC_LAY_ROOT<br>DLA_C_SPEC_PAGESET<br>DLA_C_SPEC_PAGE<br>DLA_C_SPEC_RECTO_PAGE<br>DLA_C_SPEC_VERSO_PAGE<br>DLA_C_SPEC_COMP_HEADER<br>DLA_C_SPEC_COMP_BODY_FIX<br>DLA_C_SPEC_COMP_BODY_VAR<br>DLA_C_SPEC_COL_FIXED<br>DLA_C_SPEC_COL_VARIABLE<br>DLA_C_SPEC_SNAKING_COLUMN |

Table 3-2 (cont.) Enumerated Data Types

| Enumerated data type | Meaning |
|---|---|
| dla_entity_type (cont.) | DLA_C_SPEC_SYNC_COLUMN<br>DLA_C_SPEC_BASIC_FLOAT<br>DLA_C_SPEC_COMP_FLOAT<br>DLA_C_SPEC_BASIC_COLUMN<br>DLA_C_SPEC_FOOTNOTE_AREA<br>DLA_C_SPEC_ARRNGD_CONT_FIXED<br>DLA_C_SPEC_ARRNGD_CONT_VAR<br>DLA_C_SPEC_SRCD_CONT_FIXED<br>DLA_C_SPEC_SRCD_CONT_VAR<br>DLA_C_SPEC_COMP_FIXTURE_VAR<br>DLA_C_SPEC_COMP_FIXTURE_FIXED<br>DLA_C_SPEC_BASIC_FIXTURE<br>DLA_C_SPEC_COMP_COL_FIXED<br>DLA_C_SPEC_COMP_COL_VARIABLE<br>DLA_C_SPEC_COMP_COMMON<br>DLA_C_SPEC_COMP_ARTWORK<br>DLA_C_SPEC_BASIC_HEADER<br>DLA_C_SPEC_BASIC_BODY<br>DLA_C_SPEC_GENERIC_BLOCK<br>DLA_C_SPEC_SPECIFIC_BLOCK<br>DLA_C_SPEC_FORM_AREA<br>DLA_C_SPEC_COMP_FOOTER<br>DLA_C_SPEC_BASIC_FOOTER<br>DLA_C_SPEC_TABLE_HEADER<br>DLA_C_SPEC_ENTRY_GROUP_AREA<br>DLA_C_SPEC_TABLE_AREA<br>DLA_C_SPEC_TABLE_LABEL<br>DLA_C_SPEC_COMP_TABLE_LABEL<br>DLA_C_SPEC_LABEL_COMPONENT<br>DLA_C_SPEC_ROW_AREA<br>DLA_C_SPEC_CELL<br>DLA_C_SPEC_SUBROW_GROUP<br>DLA_C_SPEC_SUBROW<br>DLA_C_SPEC_TABLE_LABEL_CONTENT<br>DLA_C_SPEC_FORM_ENTRY_AREA<br><br>DLA_C_GEN_LOG_ROOT<br>DLA_C_GEN_PASSAGE<br>DLA_C_GEN_NUM_SEG<br>DLA_C_GEN_NUMBER<br>DLA_C_GEN_TITLE<br>DLA_C_GEN_CAPTION<br>DLA_C_GEN_PARAGRAPH<br>DLA_C_GEN_PHRASE<br>DLA_C_GEN_FOOTNOTE<br>DLA_C_GEN_FOOTNOTE_NUM<br>DLA_C_GEN_FOOTNOTE_REF<br>DLA_C_GEN_FOOTNOTE_BODY<br>DLA_C_GEN_FOOTNOTE_TEXT<br>DLA_C_GEN_FIGURE<br>DLA_C_GEN_BODY_TEXT<br>DLA_C_GEN_REFERENCE<br>DLA_C_GEN_REF_CONTENT<br>DLA_C_GEN_BODY_RASTER<br>DLA_C_GEN_BODY_GEOM |

**Table 3-2 (cont.) Enumerated Data Types**

| Enumerated data type | Meaning |
|---|---|
| dla_entity_type (cont.) | DLA_C_GEN_COMMON_CONTENT<br>DLA_C_GEN_COMMON_TEXT<br>DLA_C_GEN_COMMON_RASTER<br>DLA_C_GEN_COMMON_GEOM<br>DLA_C_GEN_DESCRIPTION<br>DLA_C_GEN_ARTWORK<br>DLA_C_GEN_NUMBERED_LIST<br>DLA_C_GEN_UNNUMBERED_LIST<br>DLA_C_GEN_DEFINITION_LIST<br>DLA_C_GEN_LIST_ITEM<br>DLA_C_GEN_LIST_TERM<br>DLA_C_GEN_TABLE<br>DLA_C_GEN_ROW<br>DLA_C_GEN_TABLE_COMPONENT<br>DLA_C_GEN_ROW_COMPONENT<br>DLA_C_GEN_FORM<br>DLA_C_GEN_ENTRY_ELEMENT<br>DLA_C_GEN_ENTRY_GROUP<br>DLA_C_GEN_COMMON_REFERENCE<br>DLA_C_GEN_COMMON_NUMBER<br>DLA_C_GEN_CURRENT_INSTANCE<br>DLA_C_GEN_PAGE_NUMBER<br>DLA_C_GEN_ENTRY_TEXT<br>DLA_C_GEN_ENTRY_RASTER<br>DLA_C_GEN_ENTRY_GEOMETRIC<br>DLA_C_GEN_TABLE_NUMBER |
| dla_document_options | Document options<br><br>DLA_C_NO_OPTIONS<br>DLA_C_MODE_1 (= NO_OPTIONS)<br>DLA_C_AUTO_GEN (= NO_OPTIONS)<br>DLA_C_MODE_2<br>DLA_C_WITH_EXTERNAL<br>DLA_C_FOR_EXTERNAL<br>DLA_C_FOR_RESOURCE |
| dla_generic_doc_type | Generic document types<br><br>DLA_C_EXTERNAL_DOC<br>DLA_C_RESOURCE_DOC |
| dla_pageset_type | Pages required when creating a pageset<br><br>DLA_C_TITLE_PAGE<br>DLA_C_SINGLE_PAGE<br>DLA_C_RECTO_VERSO<br>DLA_C_TITLE_SINGLE_PAGE<br>DLA_C_TITLE_VERSO_RECTO |
| dla_link_option | Link options for specific logical entities<br><br>DLA_C_LINK_BEFORE<br>DLA_C_LINK_AFTER<br>DLA_C_LINK_AS_FIRST<br>DLA_C_LINK_AS_LAST |
| dla_child_type | The first or last child of an entity<br><br>DLA_C_CHILD_FIRST<br>DLA_C_CHILD_LAST |

**Table 3-2 (cont.) Enumerated Data Types**

| Enumerated data type | Meaning |
|---|---|
| dla_sibling_type | The first, last, next, or previous sibling |
| | DLA_C_SIB_FIRST<br>DLA_C_SIB_LAST<br>DLA_C_SIB_NEXT<br>DLA_C_SIB_PREV |
| dla_structure_type | The type of structure |
| | DLA_C_GENERIC_LAYOUT<br>DLA_C_SPECIFIC_LOGICAL<br>DLA_C_GENERIC_LOGICAL<br>DLA_C_SPECIFIC_LAYOUT |
| dla_toolkit_options | Toolkit options |
| | DLA_C_NO_OPTIONS |
| dla_property_code | Property code |
| dla_index_operation | Insert or replace an entry in a sequence |
| | DLA_C_INDEX_INSERT<br>DLA_C_INDEX_REPLACE |
| dla_property_status | Property status |
| | DLA_C_PROP_SPECIFIED<br>DLA_C_PROP_UNSPECIFIED<br>DLA_C_PROP_NULL |
| dla_derived_location | Where a property applying to an entity is located. |
| | DLA_C_DEF_ENTITY<br>DLA_C_DEF_STYLE<br>DLA_C_DEF_CLASS<br>DLA_C_DEF_CLASS_STYLE<br>DLA_C_DEF_REF_CLASS<br>DLA_C_DEF_REF_STYLE<br>DLA_C_DEF_VAL_LIST<br>DLA_C_DEF_DAP_DEF<br>DLA_C_DEF_STD_DEF |
| dla_tab_alignment | Tabulation stop alignment |
| | DLA_C_TAB_ALIGN_START<br>DLA_C_TAB_ALIGN_END<br>DLA_C_TAB_ALIGN_CENTRED<br>DLA_C_TAB_ALIGN_AROUND |
| dla_snaking_type | Type of snaking columns |
| | DLA_C_SNAKING_REP<br>DLA_C_SNAKING_SEQ |
| dla_header_footer_type | Type of header or footer |
| | DLA_C_COMP_HEAD_FIXED<br>DLA_C_COMP_HEAD_VAR<br>DLA_C_COMP_FOOT_FIXED<br>DLA_C_COMP_FOOT_VAR<br>DLA_C_BASIC_HEAD<br>DLA_C_BASIC_FOOT |

Table 3-2 (cont.) Enumerated Data Types

| Enumerated data type | Meaning |
|---|---|
| dla_conversion_type | Direction of content conversion |
| | DLA_C_CHAR_6937_TO_8859 |
| | DLA_C_CHAR_8859_TO_6937 |
| | DLA_C_RAS_TO_T4_1D |
| | DLA_C_RAS_TO_T4_2D |
| | DLA_C_RAS_TO_T6 |
| | DLA_C_RAS_TO_BIT |

## 3.2 Constants

Symbolic constants are used in two contexts. These contexts are:

- To identify the choice when a property can be represented in more than one way. These constants are listed in 3.2.1, which describes choice identifiers.
- To provide the values for many properties. These constants are listed in 3.2.2, which describes permitted values of properties.

### 3.2.1 Choice Identifiers

The choice identifiers are used when reading a document. They determine which property from a choice of properties is present. This subclause lists the choice identifiers.

1. The choice identifiers for the Document reference properties:
   DLA_C_DOC_REF
   DLA_C_EXT_DOC
   DLA_C_RES_DOC
   DLA_C_REFS_OTHER_DOCS_TY
   DLA_C_SUPERSEDED_DOCS_TY

| Meaning | Constant |
|---|---|
| Unique reference | DLA_C_REF_UNIQ |
| Descriptive reference | DLA_C_REF_DESC |

2. The choice identifiers for the Authorization property:
   DLA_C_SEC_AUTHORIZATION

| Meaning | Constant |
|---|---|
| Personal name | DLA_C_AUTH_PERSONAL_NAME |
| Organization | DLA_C_AUTH_ORGANIZATION |

3. The choice identifiers for:

   - The raster graphics content presentation property: Image dimensions
   - The geometric graphics content presentation property: Picture dimensions

| Meaning | Constant |
|---|---|
| Width controlled | DLA_C_GRA_DIM_WID |
| Height controlled | DLA_C_GRA_DIM_HGT |
| Area controlled | DLA_C_GRA_DIM_AREA |
| Automatic | DLA_C_GRA_DIM_AUTO |

4. The choice identifiers for the raster graphics content presentation property: Pel spacing.

| Meaning | Constant |
|---|---|
| Spacing (that is, length and pel spaces) | DLA_C_PEL_SPC |
| Null | DLA_C_PEL_NUL |

5. The choice identifiers for the Non-basic character features property: DLA_C_NONB_CHAR_FEAT

| Meaning | Constant |
|---|---|
| Character spacing | DLA_C_FEAT_CHR_SPC |
| Character orientation | DLA_C_FEAT_CHR_ORN |
| Character path | DLA_C_FEAT_CHR_PTH |
| Graphic character sets | DLA_C_FEAT_GR_CHR_SET |
| Graphic rendition | DLA_C_FEAT_GR_RND |
| Graphic character subrepertoire | DLA_C_FEAT_CHR_SUB_REP |
| Line progression | DLA_C_FEAT_LIN_PGR |
| Line spacing | DLA_C_FEAT_LIN_SPC |

6. The choice identifiers for the Non-basic raster feature property: DLA_C_PRES_FEAT

| Meaning | Constant |
|---|---|
| Pel spacing | DLA_C_FEAT_PEL_SPACING |
| Pel path | DLA_C_FEAT_PEL_PATH |
| Line progression | DLA_C_FEAT_LIN_PROG |
| Spacing ratio | DLA_C_FEAT_SPC_RTO |

7. The choice identifiers for the Non-basic pel spacings property: DLA_C_NONB_PEL_SPACINGS

| Meaning | Constant |
|---|---|
| Pel spacing | DLA_C_FEAT_PEL_SPC |
| Pel spacing null | DLA_C_FEAT_PEL_SPC_NUL |

The null value should not occur because null is not non-basic.

8. The choice identifiers for the Non-basic geo features property: DLA_C_GEO_FEAT

| Meaning | Constant |
|---|---|
| Geo feature | DLA_C_GEO_FEAT_TEXT_RND |
| Geo feature null | DLA_C_GEO_FEAT_NUL |

The null value should not occur because null is not non-basic.

9. The choice identifiers for the Path dimension and Orthogonal dimension properties. Not all choices are applicable to both properties and all entities. Appendix B lists the properties that can be set for each entity.

| Meaning | Constant |
|---|---|
| Rule A | DLA_C_DIM_RUL_A |
| Rule B | DLA_C_DIM_RUL_B |
| Fixed dimension | DLA_C_DIM_FIX |
| Maximum size applies | DLA_C_DIM_MAX_SIZ |

10. The choice identifiers for the Remaining pages property: DLA_C_GEN_REM_PGE

| Meaning | Constant |
|---|---|
| Single page | DLA_C_PGE_SIN |
| Recto and verso pages | DLA_C_PGE_REC_VER |

11. The choice identifiers for the Indivisibility property: DLA_C_LYD_IDV

| Meaning | Constant |
|---|---|
| Page indivisibility | DLA_C_IDV_PGE |
| No indivisibility | DLA_C_IDV_NUL |
| Area indivisibility | DLA_C_IDV_ARE |
| Stream indivisibility | DLA_C_IDV_CAT |

12. The choice identifiers for the Synchronization property: DLA_C_LYD_SYN

| Meaning | Constant |
|---|---|
| Synchronized entity | DLA_C_SYN_ENT |
| Synchronized expression | DLA_C_SYN_EXP (unsupported choice) |

13. The choice identifiers for the Same layout object property: DLA_C_LYD_SLO

| Meaning | Constant |
|---|---|
| Same page | DLA_C_SLO_PGE |
| Same stream | DLA_C_SLO_CAT |
| Same frame | DLA_C_SLO_FRAME |
| No same layout object | DLA_C_SLO_NUL |

14. The choice identifiers for the New layout object property: DLA_C_LYD_NEW_LAY_OBJ

| Meaning | Constant |
|---|---|
| New pageset | DLA_C_NEW_PGE_SET |
| Page break | DLA_C_PGE_BRK |
| New page | DLA_C_NEW_PGE |
| Column break | DLA_C_COL_BRK |
| New layout | DLA_C_NEW_LAY |

15. The choice identifiers for the Layout in cell property: DLA_C_LYD_CELL

| Meaning | Constant |
|---|---|
| By category | DLA_C_CELL_CAT |
| By entity | DLA_C_CELL_ENT |

16. The choice identifiers for the Position property:
DLA_C_POS

| Meaning | Constant |
|---|---|
| Fixed position | DLA_C_POS_FIX |
| Variable position | DLA_C_POS_VAR |

17. The choice identifiers for the Footnote format property:
DLA_C_FNOTE_FMT

| Meaning | Constant |
|---|---|
| Footnote string | DLA_C_FNOTE_CH_STR |
| Footnote string function | DLA_C_FNOTE_CH_FUN |

18. The choice identifiers for the FOD36 Reference, General and special purpose numbering prefix and suffix properties:
DLA_C_GB_REF_PRE
DLA_C_GB_REF_SUF
DLA_C_REF_CON_LOC_PRE
DLA_C_REF_CON_LOC_SUF
DLA_C_REF_CON_PRE
DLA_C_REF_CON_SUF
DLA_C_COM_REF_LOC_PRE
DLA_C_COM_REF_LOC_SUF
DLA_C_COM_REF_PRE
DLA_C_COM_REF_SUF
DLA_C_COM_NUM_PRE
DLA_C_COM_NUM_SUF
DLA_C_CUR_INS_PRE
DLA_C_CUR_INS_SUF
DLA_C_GEN_NUM_PRE
DLA_C_GEN_NUM_SUF
DLA_C_TAB_NUM_PRE
DLA_C_TAB_NUM_SUF
DLA_C_GEN_FTN_PRE
DLA_C_GEN_FTN_SUF
DLA_C_GEN_PGN_PRE
DLA_C_GEN_PGN_SUF

| Meaning | Constant |
|---|---|
| Literal string | DLA_C_PRE_SUF_STR |
| Via binding index | DLA_C_PRE_SUF_BDR |

19. The choice identifiers for the GenericBlock reference format: DLA_C_GB_REF_FMT

| Meaning | Constant |
|---|---|
| Numbers binding | DLA_C_GB_REF_FMT_NBR |
| PGnum binding | DLA_C_GB_REF_FMT_PGN |
| Numberstring binding | DLA_C_GB_REF_FMT_NBS |
| Strings binding | DLA_C_GB_REF_FMT_STR |

20. The choice identifiers for the RefContent reference format: DLA_C_REF_CON_FMT

| Meaning | Constant |
|---|---|
| Numberstring binding | DLA_C_REF_CON_CHO_NBS |
| PGnum binding | DLA_C_REF_CON_CHO_PGN |
| Numbers binding | DLA_C_REF_CON_CHO_NBR |
| Strings binding | DLA_C_REF_CON_CHO_STR |

21. The choice identifiers for the CommonRef reference format: DLA_C_COM_REF_FMT

| Meaning | Constant |
|---|---|
| Numberstring binding | DLA_C_COM_REF_CHO_NBS |
| Fnotestring binding | DLA_C_COM_REF_CHO_FTN |
| PGnum binding | DLA_C_COM_REF_CHO_PGN |
| Numbers binding | DLA_C_COM_REF_CHO_NBR |
| Strings binding | DLA_C_COM_REF_CHO_STR |

22. The choice identifiers for the CommonNumber reference format: DLA_C_COM_NUM_FMT

| Meaning | Constant |
|---|---|
| Numberstring binding | DLA_C_COM_NUM_FMT_NBS |
| Numbers binding | DLA_C_COM_NUM_FMT_NBR |

23. The choice identifiers for the ReferencedContent and CommonReference referent entity:
DLA_C_REF_CON_NBS_RFT
DLA_C_REF_CON_NBR_RFT
DLA_C_REF_CON_STR_RFT
DLA_C_COM_REF_NBS_RFT
DLA_C_COM_REF_FTN_RFT
DLA_C_COM_REF_NBR_RFT
DLA_C_COM_REF_STR_RFT
DLA_C_COM_NUM_ANY_OBJ

| Meaning | Constant |
|---|---|
| By entity type | DLA_C_REF_RFT_CHO_TYP |
| By generic entity | DLA_C_REF_RFT_CHO_CLS |

24. The choice identifiers for the PgNumbers referent entity:
DLA_C_GEN_PGN_NUM_RFT

| Meaning | Constant |
|---|---|
| By entity type | DLA_C_GEN_PGN_NUM_RFT_TYP |
| By generic entity | DLA_C_GEN_PGN_NUM_RFT_CLS |

25. The choice identifiers for the CommonRef local prefix and suffix and for CurrentInstance referent entity:
DLA_C_COM_REF_LOC_OBJ
DLA_C_CUR_INS_CUR_OBJ

| Meaning | Constant |
|---|---|
| By entity type | DLA_C_REF_RFT_CHO_TYP |
| By generic entity | DLA_C_REF_RFT_CHO_CLS |

26. The choice identifiers for the GeneralNum (Number) UseNumberStrings expression:
DLA_C_GEN_NUM_EXP_TYP

| Meaning | Constant |
|---|---|
| Literal string | DLA_C_GEN_NUM_CHO_STR |
| Via Numbers binding | DLA_C_GEN_NUM_CHO_NUM |
| ORD function | DLA_C_GEN_NUM_CHO_ORD |

27. The choice identifiers for the General footnote numbering format property: DLA_C_GEN_FTN_NUM

| Meaning | Constant |
|---|---|
| Footnote numberstring | DLA_C_GEN_FTN_CHO_IDX |
| Footnote literal string | DLA_C_GEN_FTN_CHO_STR |
| Footnote fnotestring | DLA_C_GEN_FTN_CHO_FNS |

28. The choice identifiers for the Snaking column property: DLA_C_SNA_COL

| Meaning | Constant |
|---|---|
| Repeat column variable | DLA_C_COL_VAR_REP |
| Sequence column variable | DLA_C_COL_VAR_SEQ |

29. The choice identifiers for the font properties: Owner prefix description, Owner assigned name description (these are subparameters of the Global Name structure), the Font resource name property, Design source name property and Writing mode name property.

| Meaning | Constant |
|---|---|
| Numeric | DLA_C_MESSAGE_NUMERIC |
| Printable | DLA_C_MESSAGE_PRINTABLE |
| Teletex | DLA_C_MESSAGE_TELETEX |
| Videotex | DLA_C_MESSAGE_VIDEOTEX |
| Visible | DLA_C_MESSAGE_VISIBLE |
| Ia5 | DLA_C_MESSAGE_IA5 |
| Graphic | DLA_C_MESSAGE_GRAPHIC |
| General | DLA_C_MESSAGE_GENERAL |

30. The choice identifiers for the Font family name property.

| Meaning | Constant |
|---|---|
| Numeric | DLA_C_MATCH_NUMERIC |
| Printable | DLA_C_MATCH_PRINTABLE |
| Teletex | DLA_C_MATCH_TELETEX |
| Videotex | DLA_C_MATCH_VIDEOTEX |
| Visible | DLA_C_MATCH_VISIBLE |
| Ia5 | DLA_C_MATCH_IA5 |
| Graphic | DLA_C_MATCH_GRAPHIC |
| General | DLA_C_MATCH_GENERAL |

31. The choice identifiers for the property Font local name subparameter of the Font global name structure, and for the properties Font resource name, Design source name and Writing mode name.

| Meaning | Constant |
|---|---|
| Name | DLA_C_LOCAL_NAME_NAME |
| Number | DLA_C_LOCAL_NAME_NUMBER |

32. The choice identifiers for the geometric graphics presentation property: Region of interest

| Meaning | Constant |
|---|---|
| Automatic | DLA_C_GA_REG_SPEC_AUTO |
| Rectangle | DLA_C_GA_REG_SPEC_RECT |

### 3.2.2 Permitted Values of Properties

This subclause lists the constants that represent the permitted values of properties.

1. The Document application profile property:
DLA_C_DAP

| Meaning | Constant |
|---|---|
| FOD11 | DLA_C_FOD11 |
| FOD26 | DLA_C_FOD26 |
| FOD36 | DLA_C_FOD36 |

2. The Document architecture class property:
DLA_C_DOC_ARC_CLS

| Meaning | Constant |
|---|---|
| Formatted | DLA_C_DOC_ARC_CLS_F |
| Processable | DLA_C_DOC_ARC_CLS_P |
| Formatted processable | DLA_C_DOC_ARC_CLS_FP |

3. The character content presentation properties:
Character orientation, DLA_C_CA_CHR_ORN
Character path, DLA_C_CA_CHR_PTH
The raster content presentation properties:
(Non-basic) Pel path, DLA_C_RA_PEL_PATH, DLA_C_NONB_PEL_PATH
The geometric graphic content presentation property:
Picture orientation DLA_C_GA_PIC_ORN

| Meaning | Constant |
|---|---|
| 0° | DLA_C_ANGLE_0 |
| 90° | DLA_C_ANGLE_90 |
| 180° | DLA_C_ANGLE_180 |
| 270° | DLA_C_ANGLE_270 |

4. The character content presentation property:
Line progression, DLA_C_CA_LIN_PGR
The raster content presentation properties:
(Non-basic) Line progression, DLA_C_RA_LIN_PROG, DLA_C_NONB_LINE_PROG

| Meaning | Constant |
|---|---|
| 90° | DLA_C_ANGLE_90 |
| 270° | DLA_C_ANGLE_270 |

5. The character content presentation property: Alignment
DLA_C_CA_ALN

| Meaning | Constant |
|---|---|
| Start aligned | DLA_C_CHR_ALN_SRT |
| End aligned | DLA_C_CHR_ALN_END |
| Centered | DLA_C_CHR_ALN_CEN |
| Justified | DLA_C_CHR_ALN_JUS |

6. The SGR control function, which is the value of the (non-basic) character content presentation property: Graphic rendition
DLA_C_CA_GR_RND_PAR, DLA_C_NONB_GRAPH_REND

The lowest DAP at which the rendition value applies is given, for example, (FOD11)

| Meaning | Constant | |
|---|---|---|
| Default rendition | DLA_C_REN_CANCEL | (FOD11) |
| Increased intensity | DLA_C_REN_INC_INT | (FOD11) |
| Decreased intensity | DLA_C_REN_DEC_INT | (FOD36) |
| Italicized | DLA_C_REN_ITL | (FOD11) |

| Meaning | Constant | |
|---|---|---|
| Underlined | DLA_C_REN_UNDER | (FOD11) |
| Slowly blinking | DLA_C_REN_SLOW_BLINK | (FOD36) |
| Rapidly blinking | DLA_C_REN_RAPID_BLINK | (FOD36) |
| Negative image | DLA_C_REN_NEG_IMAGE | (FOD36) |
| Crossed out | DLA_C_REN_CROSS_OUT | (FOD11) |
| Primary font | DLA_C_REN_PRIM_FONT | (FOD26) |
| First alternative font | DLA_C_REN_FRS_ALT_FONT | (FOD26) |
| Second alternative font | DLA_C_REN_SEC_ALT_FONT | (FOD26) |
| Third alternative font | DLA_C_REN_TRD_ALT_FONT | (FOD26) |
| Fourth alternative font | DLA_C_REN_FOUR_ALT_FONT | (FOD26) |
| Fifth alternative font | DLA_C_REN_FIF_ALT_FONT | (FOD26) |
| Sixth alternative font | DLA_C_REN_SIX_ALT_FONT | (FOD26) |
| Seventh alternative font | DLA_C_REN_SEV_ALT_FONT | (FOD26) |
| Eighth alternative font | DLA_C_REN_EIG_ALT_FONT | (FOD26) |
| Ninth alternative font | DLA_C_REN_NIN_ALT_FONT | (FOD26) |
| Doubly underlined | DLA_C_REN_DBL_UNDER | (FOD26) |
| Normal intensity | DLA_C_REN_NOR_INT | (FOD11) |
| Not italicized | DLA_C_REN_NOT_ITL | (FOD11) |
| Not underlined | DLA_C_REN_NOT_UNDER | (FOD11) |
| Not blinking (steady) | DLA_C_REN_STEADY | (FOD36) |
| Positive image | DLA_C_REN_POS_IMAGE | (FOD36) |
| Not crossed out | DLA_C_REN_NOT_CROSS_OUT | (FOD11) |

7. The character content presentation property: Proportional line spacing
DLA_C_CA_PRP_LIN_SPC

| Meaning | Constant |
|---|---|
| Yes | DLA_C_PLS_YES |
| No | DLA_C_PLS_NO |

8. The character content presentation property: Pairwise kerning
DLA_C_CA_PAIR_KERN

| Meaning | Constant |
|---|---|
| Yes | DLA_C_KERN_YES |
| No | DLA_C_KERN_NO |

9. The character content presentation property: Formatting indicator
DLA_C_CA_FORM_IND

| Meaning | Constant |
|---|---|
| Yes | DLA_C_FORM_YES |
| No | DLA_C_FORM_NO |

10. The character content presentation property: First line format
DLA_C_CA_FRS_LIN_FMT

| Meaning | Constant |
|---|---|
| Positive first line offset (Example 10.1 in Figure 6-10 of *ISO 8613-6*) | DLA_C_POS_FRS_LIN_OFF |
| Negative first line offset (Example 10.2 in Figure 6-10 of *ISO 8613-6*) | DLA_C_NEG_FRS_LIN_OFF |
| Item identifier type 1 (Example 10.3 in Figure 6-10 of *ISO 8613-6*) | DLA_C_ITM_ID_TYP_1 |
| Item identifier type 2 (Example 10.4 in Figure 6-10 of *ISO 8613-6*) | DLA_C_ITM_ID_TYP_2 |
| Item identifier type 3 (Example 10.5 in Figure 6-10 of *ISO 8613-6*) | DLA_C_ITM_ID_TYP_3 |
| First line format undefined | DLA_C_FRS_LIN_FMT_UNKNOWN |

11. The character content presentation property: Identifier alignment
DLA_C_CA_ITM_ALN

| Meaning | Constant |
|---|---|
| No itemization | DLA_C_ID_ALN_NO_ITM |
| Start aligned | DLA_C_ID_ALN_SRT |
| End aligned | DLA_C_ID_ALN_END |

12. The content architecture class properties:
DLA_C_CON_ARC_CLS
DLA_C_PRF_CON_ARC_CLS
DLA_C_DEF_ARC_CLS

| Meaning | Constant |
|---|---|
| Formatted character content | DLA_C_CON_F_CHR |
| Processable character content | DLA_C_CON_P_CHR |
| Formatted processable character content | DLA_C_CON_FP_CHR |
| Formatted processable raster graphic content | DLA_C_CON_FP_RAS |
| Formatted processable geometric graphic content | DLA_C_CON_FP_GEO |

13. The Aspect ratio flag subparameter of the raster graphics content presentation property: Image dimensions
DLA_C_RA_IMG_AR_RTO

The Aspect ratio flag subparameter of the geometric graphics content presentation property: Picture dimensions
DLA_C_GA_PIC_AR_RTO

| Meaning | Constant |
|---|---|
| Fixed | DLA_C_ASP_RTO_FIX |
| Variable | DLA_C_ASP_RTO_VAR |

14. The Page layout type property:
DLA_C_PGE_LAY_TYP

| Meaning | Constant |
|---|---|
| Page layout type A | DLA_C_PGE_LAY_A |
| Page layout type B | DLA_C_PGE_LAY_B |
| Page layout type C | DLA_C_PGE_LAY_C |
| Page layout type D | DLA_C_PGE_LAY_D |

For FOD36 there is an additional set of constants:

| Meaning | Constant |
|---|---|
| Page layout type A2 | DLA_C_PGE_LAY_A2 |
| Page layout type A3 | DLA_C_PGE_LAY_A3 |
| Page layout type B1 | DLA_C_PGE_LAY_B1 |
| Page layout type B3 | DLA_C_PGE_LAY_B3 |
| Page layout type C1 | DLA_C_PGE_LAY_C1 |

Notes:

Specifying the value DLA_C_PGE_LAY_A2 causes the layout path of the body area to be set to 270° and the layout path of the header and footer area to be set to 0°.

Specifying the value DLA_C_PGE_LAY_A3 causes the layout path of the body area to be set to 270° and the layout path of the header and footer area to be set to 180°.

Specifying the value DLA_C_PGE_LAY_B1 causes the layout path of the body area to be set to 0° and the layout path of the header and footer area to be set to 180°.

Specifying the value DLA_C_PGE_LAY_B3 causes the layout path of the body area to be set to 0° and the layout path of the header and footer area to be set to 0°.

Specifying the value DLA_C_PGE_LAY_C1 causes the layout path of the body area to be set to 180° and the layout path of the header and footer area to be set to 0°.

15. The Page orientation property:
DLA_C_PGE_ORIENTN

| Meaning | Constant |
|---|---|
| Landscape | DLA_C_PGE_ORIENT_LAND |
| Portrait (and square) | DLA_C_PGE_ORIENT_PORT |

16. The Layout direction property:
DLA_C_LAY_DIR

| Meaning | Constant |
|---|---|
| Layout path 0/270° | DLA_C_NORMAL_DIRECTION |
| Layout path 180/90° | DLA_C_REVERSE_DIRECTION |

DLA_C_NORMAL_DIRECTION is layout path 0° for Page layout type A and 270° for Page layout type B. DLA_C_REVERSE_DIRECTION is layout path 180° for Page layout type A and 90° for Page layout type B.

17. The Medium size property:
DLA_C_PRS_MED_SIZ
DLA_C_NONB_MED_SIZ

| Meaning | Constant |
|---|---|
| ISO A5 landscape | DLA_C_SIZ_ISO_A5_L |
| ISO A4 landscape | DLA_C_SIZ_ISO_A4_L |
| ISO A3 landscape | DLA_C_SIZ_ISO_A3_L |
| ISO A2 landscape | DLA_C_SIZ_ISO_A2_L |
| ISO A1 landscape | DLA_C_SIZ_ISO_A1_L |
| ISO A0 landscape | DLA_C_SIZ_ISO_A0_L |
| ANSI legal landscape | DLA_C_SIZ_ANS_LEG_L |
| ANSI A landscape | DLA_C_SIZ_ANS_A_L |
| ANSI B landscape | DLA_C_SIZ_ANS_B_L |
| ANSI C landscape | DLA_C_SIZ_ANS_C_L |
| ANSI D landscape | DLA_C_SIZ_ANS_D_L |
| ANSI E landscape | DLA_C_SIZ_ANS_E_L |
| ANSI F landscape | DLA_C_SIZ_ANS_F_L |
| Japan legal landscape | DLA_C_SIZ_JAP_LEG_L |
| Japan letter landscape | DLA_C_SIZ_JAP_LET_L |
| ISO A5 portrait | DLA_C_SIZ_ISO_A5_P |
| ISO A4 portrait | DLA_C_SIZ_ISO_A4_P |
| ISO A3 portrait | DLA_C_SIZ_ISO_A3_P |
| ISO A2 portrait | DLA_C_SIZ_ISO_A2_P |
| ISO A1 portrait | DLA_C_SIZ_ISO_A1_P |
| ISO A0 portrait | DLA_C_SIZ_ISO_A0_P |

| Meaning | Constant |
|---|---|
| ANSI legal portrait | DLA_C_SIZ_ANS_LEG_P |
| ANSI A portrait | DLA_C_SIZ_ANS_A_P |
| ANSI B portrait | DLA_C_SIZ_ANS_B_P |
| ANSI C portrait | DLA_C_SIZ_ANS_C_P |
| ANSI D portrait | DLA_C_SIZ_ANS_D_P |
| ANSI E portrait | DLA_C_SIZ_ANS_E_P |
| ANSI F portrait | DLA_C_SIZ_ANS_F_P |
| Japan legal portrait | DLA_C_SIZ_JAP_LEG_P |
| Japan letter portrait | DLA_C_SIZ_JAP_LET_P |

18. The Medium side property:
DLA_C_MED_SID
DLA_C_NONB_MED_SID

| Meaning | Constant |
|---|---|
| Unspecified | DLA_C_SID_UNS |
| Recto | DLA_C_SID_REC |
| Verso | DLA_C_SID_VRS |

19. The Line type subparameters of the Border properties:
DLA_C_BDR_LFT_TY
DLA_C_NONB_BORDER_LFT_TY
DLA_C_BDR_RGT_TY
DLA_C_NONB_BORDER_RGT_TY
DLA_C_BDR_TRL_TY
DLA_C_NONB_BORDER_TRL_TY
DLA_C_BDR_LDG_TY
DLA_C_NONB_BORDER_LDG_TY

| Meaning | Constant |
|---|---|
| Invisible | DLA_C_LIN_TY_INV |
| Solid | DLA_C_LIN_TY_SOL |
| Dashed | DLA_C_LIN_TY_DSH |
| Dot | DLA_C_LIN_TY_DOT |
| Dash-dot | DLA_C_LIN_TY_DSH_DOT |
| Dash-dot-dot | DLA_C_LIN_TY_DSH_DOT_DOT |

20. The Colour property:
DLA_C_COLOUR

| Meaning | Constant |
|---|---|
| Colourless | DLA_C_COL_COLOURLESS |
| White | DLA_C_COL_WHITE |

21. The Protection property:
DLA_C_PROTECTION

| Meaning | Constant |
|---|---|
| Protected | DLA_C_PROT_PROTECTED |
| Unprotected | DLA_C_PROT_UNPROTECTED |

22. The Transparency property:
DLA_C_TRANSPARENCY

| Meaning | Constant |
|---|---|
| Transparent | DLA_C_TRANS_TRANSPARENT |
| Opaque | DLA_C_TRANS_OPAQUE |

23. The Type of coding property:
DLA_C_CP_TY_COD

| Meaning | Constant |
|---|---|
| Character encoding | DLA_C_COD_CHR |
| Geometric encoding | DLA_C_COD_GEO |
| T.6 encoding | DLA_C_COD_RST_T6 |
| T.4 one dimensional encoding | DLA_C_COD_RST_T4_1D |
| T.4 two dimensional encoding | DLA_C_COD_RST_T4_2D |
| Bitmap encoding | DLA_C_COD_RST_BIT |

24. The Compression properties:
DLA_C_CP_RST_CMP and DLA_C_NONB_COMPRESS

| Meaning | Constant |
|---|---|
| Uncompressed | DLA_C_UNCOMPRESS |
| Compressed | DLA_C_COMPRESS |

25. The string function properties:
Footnote string function
DLA_C_FNOTE_STR_FUN
Page number string function
DLA_C_PGE_NUM_STR_FUN
Segment number string function
DLA_C_SEG_NUM_STR_FUN
Other (DAP Level 3) Numbering string function
DLA_C_COM_NUM_NBR_FUN
DLA_C_COM_REF_NBR_FUN
DLA_C_COM_REF_PGN_FUN
DLA_C_GB_REF_NBR_FUN
DLA_C_GB_REF_PGN_FUN
DLA_C_GEN_NUM_STR_FUN
DLA_C_GEN_PGN_NUM_FUN
DLA_C_REF_CON_NBR_FUN
DLA_C_REF_CON_PGN_FUN

| Meaning | Constant |
|---|---|
| MK_STR | DLA_C_MK_STR |
| U_ALPHA | DLA_C_U_ALPHA |
| L_ALPHA | DLA_C_L_ALPHA |
| U_ROM | DLA_C_U_ROM |
| L_ROM | DLA_C_L_ROM |

26. The Expression type parameter of the Segment number string property:
DLA_C_SEG_NUM_EXP_TYP

| Meaning | Constant |
|---|---|
| Hierarchic | DLA_C_SEG_NUM_HIER |
| Simple | DLA_C_SEG_NUM_SIMPLE |

27. The Strings required parameter of the Segment number format property:
DLA_C_SEG_NUM_STR_REQ

| Meaning | Constant |
|---|---|
| Prefix only | DLA_C_PRE_ONLY |
| Suffix only | DLA_C_SUF_ONLY |
| Prefix and suffix | DLA_C_PRE_SUF |

28. The Current object type properties:
DLA_C_COM_REF_NBR_TYP
DLA_C_COM_REF_NBS_TYP
DLA_C_COM_REF_STR_TYP
DLA_C_COM_NUM_OBJ_TYP
DLA_C_CUR_INS_OBJ_TYP

| Meaning | Constant |
|---|---|
| Current instance composite logical object | DLA_C_CUR_INS_COMP_LOG |
| Current instance frame | DLA_C_CUR_INS_FRAME |
| Current instance page | DLA_C_CUR_INS_PAGE |

29. The Common ref to footnote object type property:
DLA_C_COM_REF_FTN_TYP

| Meaning | Constant |
|---|---|
| Current instance composite logical object | DLA_C_CUR_INS_COMP_LOG |

30. The Current object type properties:
DLA_C_COM_REF_LOC_TYP
DLA_C_GEN_PGN_NUM_TYP
DLA_C_REF_CON_NBR_TYP
DLA_C_REF_CON_NBS_TYP
DLA_C_REF_CON_STR_TYP

| Meaning | Constant |
|---|---|
| Current Instance Frame | DLA_C_CUR_INS_FRAME |
| Current Instance Page | DLA_C_CUR_INS_PAGE |

31. The General page numbering current object type properties:
DLA_C_GEN_PGN_NUM_TYP

| Meaning | Constant |
|---|---|
| Current Instance Frame | DLA_C_CUR_INS_FRAME |
| Current Instance Page | DLA_C_CUR_INS_PAGE |

32. The General page numbering reference expression type properties:
DLA_C_GEN_PGN_EXP_TYP

| Meaning | Constant |
|---|---|
| NumberString reference | DLA_C_EXP_NUMBERSTRING |
| Numbers reference | DLA_C_EXP_NUMBERS |
| PGnum reference | DLA_C_EXP_PG_NUM |

33. The superior property within RefContent, CommonRef, CommonNumber, CurrentInstance:
DLA_C_REF_CON_SUP
DLA_C_COM_REF_SUP
DLA_C_COM_NUM_SUP
DLA_C_CUR_INS_SUP

| Meaning | Constant |
|---|---|
| Superior object | DLA_C_REF_TO_SUP |
| Not superior (current) | DLA_C_NOT_REF_TO_SUP |

34. The Code area argument in the functions: dla_append_char_set, dla_get_nth_char_set, dla_get_spec_nth_char_set, dla_get_index_char_set.

| Meaning | Constant |
|---|---|
| G0 | DLA_C_G0 |
| G1 | DLA_C_G1 |
| G2 | DLA_C_G2 |
| G3 | DLA_C_G3 |

35. The Graphic character set type argument in the functions: dla_append_char_set, dla_get_nth_char_set, dla_get_spec_nth_char_set, dla_get_index_char_set.

| Meaning | Constant |
|---|---|
| 94-character single-byte set | DLA_C_SIN_94 |
| 96-character single-byte set | DLA_C_SIN_96 |
| 94-character multi-byte set | DLA_C_MUL_94 |
| 96-character multi-byte set | DLA_C_MUL_96 |

36. The Concatenation property:
DLA_C_LYD_CCT

| Meaning | Constant |
|---|---|
| Nonconcatenated | DLA_C_NON_CONCAT |
| Concatenated | DLA_C_CONCAT |

37. The Block alignment property:
DLA_C_LYD_BLK_ALN

| Meaning | Constant |
|---|---|
| Left hand aligned | DLA_C_BLK_ALN_LFT |
| Right hand aligned | DLA_C_BLK_ALN_RGT |
| Centered | DLA_C_BLK_ALN_CEN |
| Null | DLA_C_BLK_ALN_NUL |

38. The Alignment subparameter of Variable position property:
DLA_C_VAR_ALN

| Meaning | Constant |
|---|---|
| Right hand aligned | DLA_C_POS_ALN_RGT |
| Left hand aligned | DLA_C_POS_ALN_LFT |
| Centered | DLA_C_POS_ALN_CEN |

39. The Page indivisibility, Page break and Same layout object properties:
DLA_C_LYD_IDV_PGE, DLA_C_LYD_PGE_BRK and DLA_C_LYD_SLO_PAGE

| Meaning | Constant |
|---|---|
| Per page | DLA_C_LAY_TY_PGE |

40. The font property: Posture code
DLA_C_FTR_POSTURE

| Meaning | Constant |
|---|---|
| Upright | DLA_C_POSTURE_UPRIGHT |
| Italic forward | DLA_C_POSTURE_ITAL_FORW |

41. The font property: Weight code
DLA_C_FTR_WEIGHT

| Meaning | Constant |
|---|---|
| Weight light | DLA_C_WEIGHT_LIGHT |
| Weight medium | DLA_C_WEIGHT_MEDIUM |
| Weight bold | DLA_C_WEIGHT_BOLD |

42. The font property: Proportionate width code
DLA_C_FTR_PROPWIDTH

| Meaning | Constant |
|---|---|
| Width undefined | DLA_C_WIDTH_UNDEFINED |
| Width ultra cond | DLA_C_WIDTH_ULTRA_COND |
| Width extra cond | DLA_C_WIDTH_EXTRA_COND |
| Width cond | DLA_C_WIDTH_COND |
| Width semi cond | DLA_C_WIDTH_SEMI_COND |
| Width medium | DLA_C_WIDTH_MEDIUM |
| Width semi expand | DLA_C_WIDTH_SEMI_EXPAND |
| Width expand | DLA_C_WIDTH_EXPAND |
| Width extra expand | DLA_C_WIDTH_EXTRA_EXPAND |
| Width ultra expand | DLA_C_WIDTH_ULTRA_EXPAND |

43. The font property: Structure code
DLA_C_FTR_STRUCTURE

| Meaning | Constant |
|---|---|
| Structure code undefined | DLA_C_STRUCT_UNDEFINED |
| Structure code solid | DLA_C_STRUCT_SOLID |
| Structure code outline | DLA_C_STRUCT_OUTLINE |

44. The font property: Escapement class
DLA_C_WRM_ESC_CLASS

| Meaning | Constant |
|---|---|
| Escapement class undefined | DLA_C_ESC_CLASS_UNDEFINED |
| Escapement class monospace | DLA_C_ESC_CLASS_MONOSPACE |
| Escapement class proportional | DLA_C_ESC_CLASS_PROPORTIONAL |

## 3.3 Error Handling

Functions are in two groups as regards error return status. The functions that operate on the document and on the toolkit initialization are in a group that returns an error status through the returning of a status object. The other group of functions are the status object interpretation functions. These do not return a success or failure indication.

There are no warning responses from function calls. A function has failed if it returns a status other than success, in which case no operation has been performed. The values of all output or write arguments to that function are undefined.

The function dla_failed is used to test whether a status object represents success or failure. If this test indicates failure, the functions dla_error_class and dla_error_number can then be used to obtain further information about the failure. In the description text for functions that return a status, it is assumed that dla_error_number is used to get the error number, and it is this number that is listed. All status codes are prefixed with DLA_E.

The status objects have an indefinite lifetime, but as they occupy no resource there is no function to release the objects.

The DAP API provides special purpose error functions to give information about the possibly complex failure modes of some functions. These special purpose functions are:

dla_get_construct_error (after failure in dla_write_document)
dla_get_fix_error (after failure in dla_fix_properties)
dla_get_interp_error (after failure in dla_read_document)
dla_get_ola_read_error (after failure in dla_read_document or in
dla_read_generic_doc)
dla_get_ola_write_error (after failure in dla_write_document)

The last two functions are only appropriate where dla_error_class or dla_error_number indicate a failure in the corresponding ODA API ODIF handling function.

The classes of error returned by the function dla_error_class are:

DLA_C_UNRECOV_ERROR
DLA_C_RECOV_ERROR
DLA_C_INTERP_ERROR
DLA_C_CONSTRUCT_ERROR
DLA_C_OLA_UNRECOV_ERROR
DLA_C_OLA_RECOV_ERROR

The application must not attempt to continue after unrecoverable errors. Unrecoverable errors can be returned by the ODA API, for example. DLA_E_MEM_FAIL and DLA_E_ODA_INTERNAL_ERROR, or by the DAP API, for example DLA_E_INTERNAL_ERROR.

The special classes DLA_C_INTERP_ERROR and DLA_C_CONSTRUCT_ERROR are associated respectively with the functions dla_read_document and dla_write_document. They indicate that the functions dla_get_interp_error and dla_get_construct_error can be used to discover additional information about the reason for failure. Examples of this usage can be found in 2.2.4. which describes toolkit and document management operations.

Status codes that can be returned by the individual functions of the DAP API are listed in 3.4, which describes each of the functions.

Generally applicable status codes that can be returned by the DAP API are given in the following list:

**[DLA_E_REQ_ARGS_NOTSPEC]**
Not all arguments were correctly specified.

**[DLA_E_MEM_FAIL]**
An error occurred when allocating or deallocating memory. This is an unrecoverable error. The application must not attempt to continue.

**[DLA_E_INTERNAL_ERROR]**
An error has occurred in the DAP Level API toolkit. This is an unrecoverable error. The application must not attempt to continue.

**[DLA_E_ODA_INTERNAL_ERROR]**
An error has occurred in the ODA Level API toolkit. This is an unrecoverable error. The application must not attempt to continue.

**[DLA_E_INV_ENTITY]**
The entity does not exist.

**[DLA_E_NO_DAP_LEVEL]**
An error occurred determining DAP level of document.

**[DLA_E_INV_PROP_OPER]**
The requested operation is not valid for this property.

**[DLA_E_NOT_A_PROPERTY_CODE]**
The property code argument does not represent any property.

**[DLA_E_INV_CODE]**
The property code does not apply to the entity.

**[DLA_E_PROP_NOT_IN_DAP]**
The property code is not valid for the entity type of the entity_handle at the DAP level of the document, but would be valid if the DAP level of the document were different.

**[DLA_E_DOC_READ_ONLY]**
The document is read only. This includes generic and resource documents.

**[DLA_E_INV_MODE]**
The writing mode of this document is not appropriate for this function.

**[DLA_E_INV_ENTITY_TYPE]**
The specified entity handle is not valid for the specified property.

**[DLA_E_INV_ENTITY_OPER]**
The requested operation is not valid for an entity of this type.

**[DLA_E_DAP_LEVEL]**
The DAP level specified is not supported.

**[DLA_E_INV_OP_AT_DAP_LEVEL]**
The requested operation is not valid at the current DAP level.