

Programming Services Volume 2
Data Services

Program Services Volume II

Chapter 1

Introduction

Copyright (c) AccuSoft Corporation, 2004. All rights reserved.

Chapter 1 - Introduction

A. Overview of Program Services

VisiQuest *Program Services* is a large group of libraries that are layered to provide the software developer with a variety of programming interfaces that trade off reduced complexity against detailed control. Program Services consists of three categories: Foundation Services, Data Services, and GUI & Visualization Services. Each Program Services category is comprised of one or more distinct libraries.

While this volume deals exclusively with Data Services, an overview of Program Services as a whole follows in order to provide a context for understanding the role of Foundation Services.

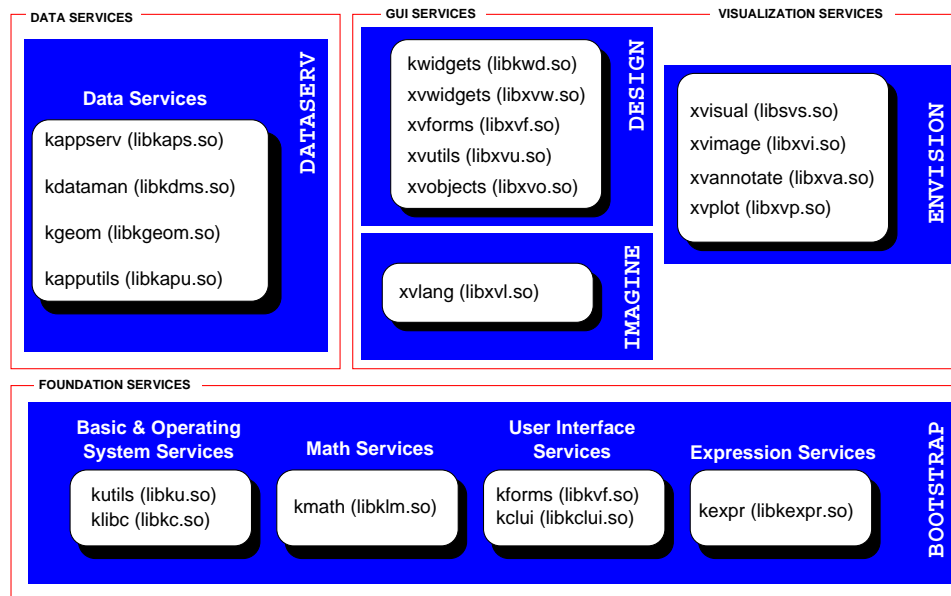


Figure 1: Program Services is comprised of libraries from the various VisiQuest toolboxes, bootstrap, devel, dataserv, design, imagine, and envision. Foundation Services is part of the bootstrap toolbox, and includes the *klibc*, *kutils*, *klibm*, and *kexpr* libraries. Data Services is provided in the *dataserv* toolbox, and is made up of the *kappserv*, *kapputils*, *kdataaccess*, *kdataman*, *kjpg*, *kdatafmt*, and *kgeom* libraries. GUI and Visualization Services is in the *design*, *imagine*, and *envision* toolboxes, and includes the *kwidgts*, *xvforms*, *xvutils*, *xvwidgts*, *xvobjects*, *xvannotate*, *xvgraphics*, *xvimage*, *xvplot*, *xvisual*, *klang*, and *xvlang* libraries.

This volume (Volume II) deals exclusively with Data Services. Volumes I and III deal with Foundation Services and GUI & Visualization Services, respectively.

B. Introduction to Data Services

Data Services consists of a collection of libraries that, together, comprise a powerful system for accessing and manipulating data. The objective of *Data Services* is to provide the application programmer with the ability to access and operate on data independent of its file format or its physical characteristics, such as size or data type. *Data Services* is designed to address the needs of a large number of application domains including image processing, signal processing, geometry visualization, and numerical analysis.

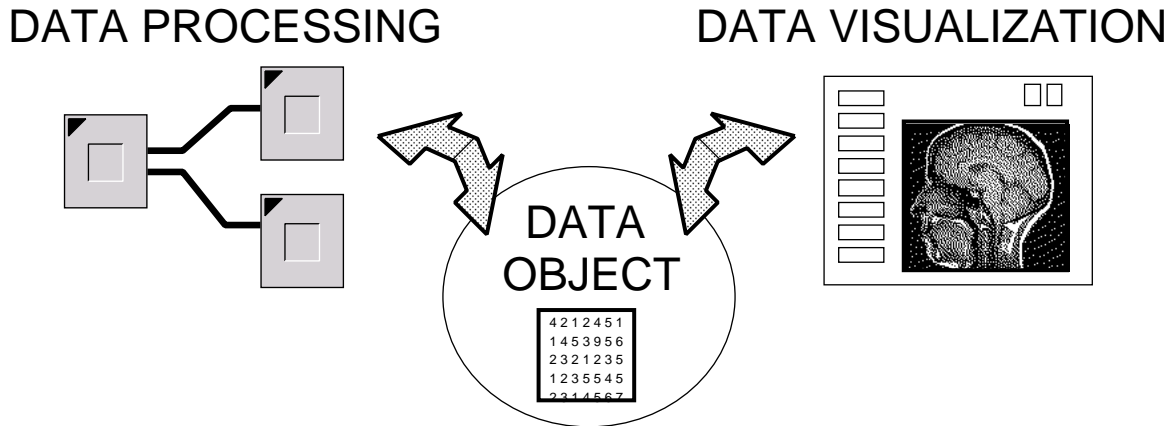


Figure 2: *Data Services* implements a powerful and abstract *data object*. This data object is used by all VisiQuest data processing and data visualization programs.

The *Data Services Application Programming Interface (API)* consists of a set of simple library functions that provide access to an abstract data object. This API allows you to store and retrieve data from the data object and to access characteristics of the data without having to worry about complicated data structures or intricate file handling. This API encapsulates extensive functionality that efficiently handles data access and presentation. This allows you to concentrate on the details of implementing your specific algorithm rather than worrying about how to access the data on which the algorithm is operating.

Many different application domains are able to utilize *Data Services*. Each domain performs all data access through the *Data Services API*. Data is interpreted according to the data model dictated by the domain. *Data Services* has a series of data models available, each of which is designed to meet the needs of a single domain or family of domains. The most powerful of these is the polymorphic data model, which provides consistent interpretation of data across many diverse domains. A geometry data model and a color data model are also available.

Data presentation routines, embedded into the *Data Services API*, handle casting, padding, resizing, scaling, and normalizing data. Data can be easily presented in whatever form is most convenient.

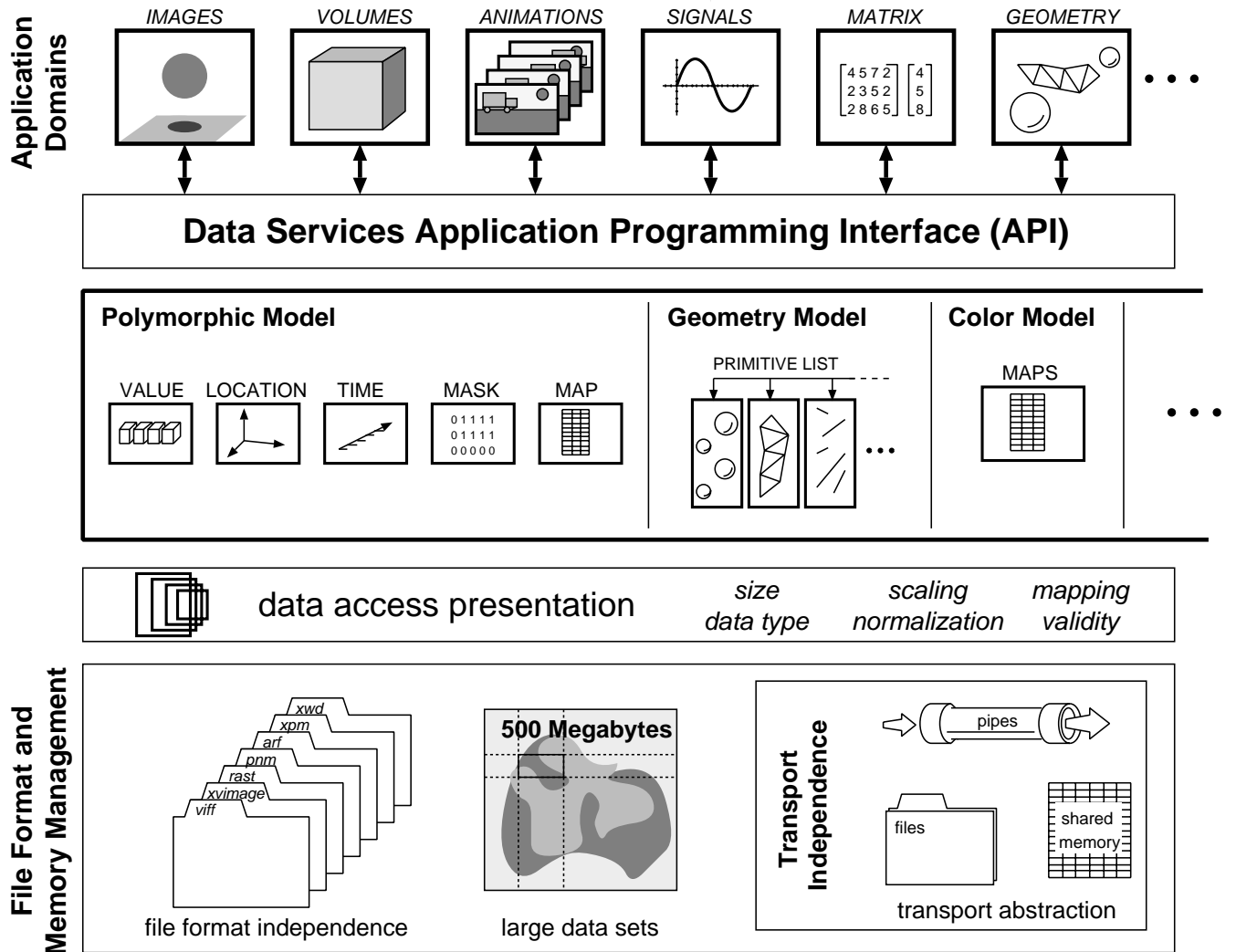


Figure 3: Many diverse application programs can be written to use Data Services. A powerful polymorphic data model ensures consistent data interpretation across the diverse domains. The complexity of handling data presentation is built into Data Services along with the ability to deal with large data sets and numerous file formats. The underlying VisiQuest transport abstraction provides Data Services with transport independence.

At the lowest level of Data Services is support for reading and writing several data file formats as well as a memory management system for accessing very large data sets. The entire system is built on the VisiQuest transport abstraction; data objects can be accessed independent of their underlying transport, whether it be a file, pipe or shared memory. The functionality provided with Data Services empowers you to write highly versatile and robust applications with a minimal amount of effort.

C. Application Programming Interface (API)

The Application Programming Interface (API) in Data Services is centered around an abstract data object made available via the data type *kobject*. You declare a *kobject* just as you would any other variable. Once declared, you can then open the object as either an input or output object, or create the object as a temporary object. After that, you can access the object with a set of application-specific function calls. Access to the object is done through primitives and attributes. †

Primitives are used to access *data* within the data object. Data is stored into the object and retrieved from the object using `put_data` and `get_data` function calls. The primitive specified with each of these calls determines the amount of data being accessed as well as where in the overall data set that data is located. ¹

Attributes are used to access *meta-data* within the data object. Meta-data is a term used loosely to cover characteristics of the data such as size and data type as well as auxiliary information such as the date or a comment. Additionally, meta-data refers to presentation information such as scaling factor or normalization range. Attributes are assigned to and retrieved from an object using `set_attribute` and `get_attribute` function calls. Functions also exist for comparing attributes of two objects, copying attributes from one object to another, and printing attributes from an object.

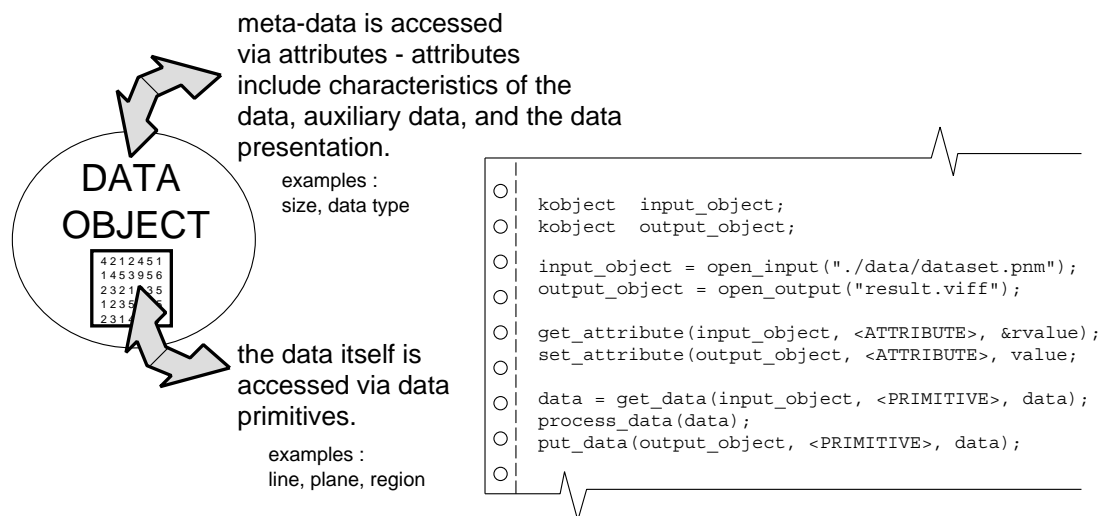


Figure 4: Data is contained within an abstract data object. This object is available for programming via the abstract data type *kobject*. The *kobject* attributes are accessed via 'get' and 'set' attribute routines and the *kobject* data is accessed via 'get' and 'put' data routines. The pseudo-code illustrates in general terms how a data processing routine utilizes Data Services. An input and output object are opened. Relevant attributes are transferred from the input to the output. Data is then retrieved from the input, processed, and finally, stored in the output.

¹ The purpose of the *kobject* data type is to hide the data structure used by Data Services from the calling application. The calling application should not change, manipulate or even see the contents of the underlying data structure; thus, the use of the *kobject*. This technique is used by several different libraries in VisiQuest system for the same reason. As such, depending on context, the *kobject* in question may be hiding different data structures. For example, the *kobject* is used by the *kutils* library to hide the data structure used for data transports, and by the *kcms* library to hide the data structure used for software objects. The *xvisual* and *xwidgets* libraries do a similar thing with the *xvobject* data type. By convention, abstract data types that are hidden from the calling application are called "kobject" if they are *not* related to visual display, "xvobject" if they are.

The primitives and attributes vary according to the data model that is used. Each data model has its own set of primitives and attributes. The specific primitives and attributes for each data model will be covered in depth in later chapters. For now, it is sufficient to understand that data objects contain data, which can be accessed via primitives, and meta-data, which can be accessed via attributes.

D. Overview of the Application Data Services

The upper level of Data Services is organized into a series of application-specific services, each with its own data model. Each data model covers the needs of either a specific domain or those of a number of similar domains. Note that even though the data models of each application service differ, the underlying philosophy, design and Application Data Services of every service is similar. This means that once you've learned one application service, you can easily learn the other application services simply by understanding their data models.

There are currently three application Data Services: *Polymorphic Services*, *Geometry Services*, and *Color Services*. Polymorphic Services is designed to cover the majority of application-domains; the polymorphic data model can store anything from signals to images and from animations to volumes. Geometry Services is designed to cover the specific needs of the geometry domain; the geometry model provides a range of geometric primitives such as triangles and spheres, in addition to a number of volumetric primitives. Color Services is an extension to Polymorphic Services with very specific functionality relating to colormaps.

If you are working with data that is raster-based in nature, consisting of discrete points in space and time, then you should use Polymorphic Data Services. Polymorphic Data Services is designed for storing up to five-dimensional data, meaning it is ideally suited for applications that need to access *signals*, *images*, *matrices*, *volumes*, or *animations*. Explicit spatial and temporal information can also be stored to position the data in space and time. This flexibility allows elevation data, for example, to be stored with a time series of registered satellite images.

If you are working with data which is vector-based in nature, consisting of geometric shapes in space, then you should use geometry data services. Geometry Data Services is designed for storing geometric primitives such as lines, triangles and spheres, meaning it is ideally suited for visualization and annotation applications. Geometry Data Services can be used for storing data such as a *road map* or an *isosurface*.

Finally, if you need to store extra color information, regardless of the nature of the data, you should use Color Data Services. Color Data Services is designed to work in conjunction with Polymorphic Data Services and Geometry Data Services by storing auxiliary color information and by generating and manipulating specialized colormaps for use with mapped data.

Please note that the data models of these services overlap wherever possible. This overlap allows processing routines written to one service to operate transparently on data from the other services. For example, a colormap generated with Color Services can be utilized directly by geometry stored with Geometry Services. The following sections will provide a brief overview of the application services. Each service will be covered in detail in later chapters.

D.1. Polymorphic Data Services

Polymorphic Data Services is the most powerful of all the application data services. The *polymorphic data model* implemented by this service is designed to encompass many application domains. This model can be used to represent data for application domains as diverse as image processing, volume processing, signal processing, computer vision and numerical analysis. By capitalizing on the commonality of data interpretation across these different domains, the polymorphic model facilitates interoperability of data manipulation routines. In other words, a processing routine written with Polymorphic Services will be able to process data objects containing anything from signals to images and from volumes to animations.

This section of the manual provides a general explanation of the polymorphic data model, as well as some examples of how data sets from different processing domains are stored in the model. You can find specific details about this model in *Chapter 2, Polymorphic Data Services* of this volume.

D.1.1. Polymorphic Data Model

The *polymorphic data model* is based on the premise that data sets are usually acquired from real-world phenomena or generated to model the same. As such, the polymorphic model consists of data that exists in three-dimensional space and one-dimensional time. You can picture the model most easily as a time-series of volumes in space. This time-series of volumes is represented by five different data segments. Each segment of data has a specific meaning dictating how it should be interpreted. Specifically, these five segments are *value*, *location*, *time*, *mask* and *map*. All of these segments are optional; a data object may contain any combination of them and still conform to the polymorphic model.

The *value* segment is the primary data segment, consisting of data element vectors organized *implicitly* into a time-series of volumes. The value data may be given *explicit* positioning in space and time with the *location* and *time* segments. The remaining two segments are provided for convenience. The *mask* segment is used to mark the validity of each point of value data. The *map* segment is provided as an extension to the value data; the value data can be used to index into the map data.

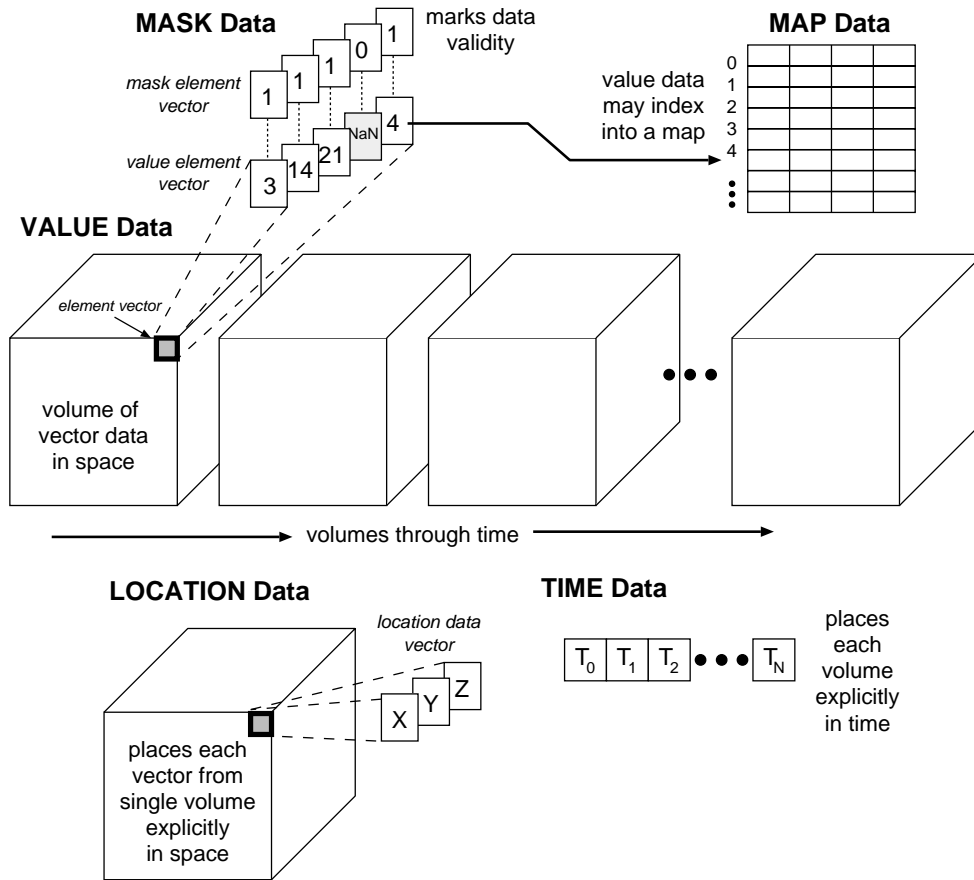


Figure 5: An overview of the Polymorphic Data Model. The polymorphic model consists of five data segments, each segment serving a specific purpose. The *value* segment consists of data element vectors organized into a time-series of volumes. The volume of value data can be given explicit locations in space with the **location** segment; one location vector is provided for each value vector in a single volume. The volumes of value data can be given explicit locations in time with the *time* segment; a time-stamp may be given for each volume in time. A *mask* segment is available for marking value data validity. A *map* segment is also provided; the value data can be used to index into the map data.

D.1.2. Value Data

The **value data** segment is the primary storage segment in the polymorphic data model. Most of the data manipulation routines are specifically geared toward processing the data stored in this segment. In an imaging context, the individual pixel RGB values would be stored here. In a signal context, regularly sampled signal amplitudes would be stored here.

The value segment consists of a time-series of volumes where each volume is composed of element vectors. Each element vector is composed of a number of value points. The size of the value segment is determined by the *width*, *height*, and *depth* of the volume, by the number of volumes through *time*, and by the number of points in the *element* vector.

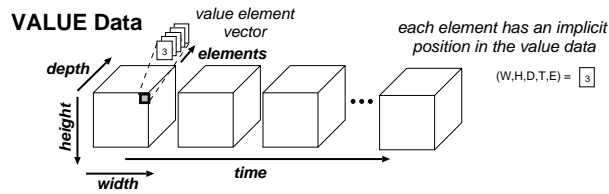


Figure 6: Polymorphic Value Data. The value segment of the polymorphic model is best pictured as a time-series of volumes. Each volume consists of element vectors oriented implicitly along the *width*, *height*, *depth*, *time*, and *elements* dimensions. Each element vector can be indexed by a four-dimensional designator while each specific value comprising the element vector can be indexed by a five-dimensional designator.

D.1.3. Location Data

The value element vectors in the value segment are stored implicitly in a regularly gridded fashion. Explicit location information can be added using the *location* segment. If the value data is irregularly sampled in space, the explicit location of each sample can be stored here. Specifically, the information stored in this segment serves to explicitly position the value data in explicit space.

The location segment consists of a volume of location vectors. The *width*, *height*, and *depth* of the volume are identical to the volume size of the value segment. Different location grid types are supported. A *curvilinear* grid allows for explicit locations to be specified for each vector in the value data. A **rectilinear** grid allows for explicit locations to be given for the *width*, *height*, and *depth* axes. A *uniform* grid allows for explicit location corner markers to be specified. Note that the location data only explicitly positions a single volume; the position then holds for each volume through time.

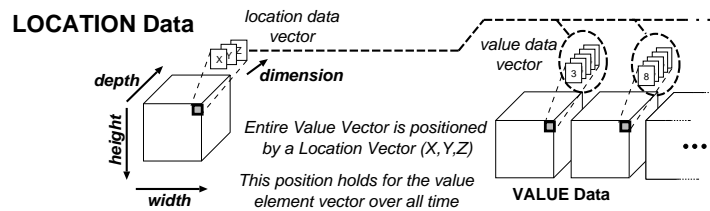


Figure 7: Polymorphic Location Data. The location segment of the polymorphic model is used to explicitly position the volume vectors in space. The location segment consists of a volume of location vectors; the *width*, *height*, and *depth* of this volume is shared from the value segment. The location vector is of size *dimension*.

D.1.4. Time Data

Explicit time information can be added using the *time* segment. If each volume of value data is irregularly sampled in time, an explicit timestamp for each volume can be stored here. This is useful in animations where each frame of the animation occurs at a different time.

The time segment consists of a linear array of timestamps. The number of timestamps matches the *time* size of the value segment.

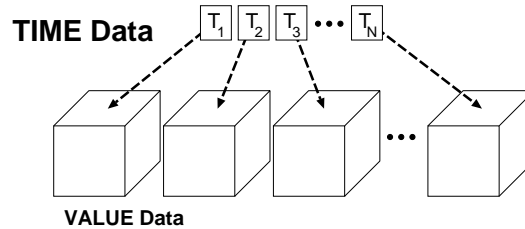


Figure 8: Polymorphic Time Data. The time segment of the polymorphic model is used to explicitly position the value volumes in time. The location segment consists of a linear array of time stamps; the number of timestamps is equivalent to the *time* size of the value segment.

D.1.5. Mask Data

The *mask* segment is available for flagging invalid values in the value segment. If a processing routine produces values, such as *NaN* or *Infinity*, these values can be flagged in the mask data so that later routines can avoid processing them. A mask point of *zero* is used to mark *invalid* value points, while a mask point of *one* is used to mark *valid* value points.

The mask segment identically mirrors the value segment in size; there is one mask point for each value point. Thus, a value in any given element vector at any given location or time can be marked as invalid.

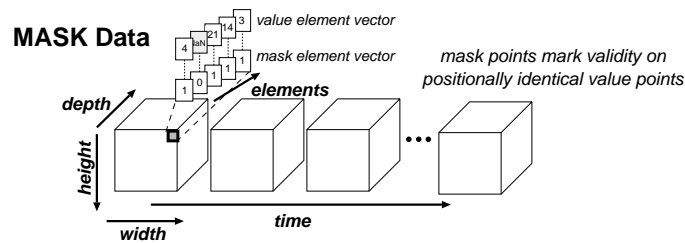


Figure 9: Polymorphic Mask Data. The mask segment of the polymorphic model is used to mark data validity of the value points. The mask segment is exactly the same size as the value segment.

D.1.6. Map Data

In cases where the value data contains redundant vectors that are duplicated in different positions within the volume, the **map** segment may be used. The value vectors are replaced with values which index into the map; the map then contains the actual data vectors. In this sense, the map is an extension of the value segment.

The map segment consists of a number of width-height planes. The values from the value segment map into the *map height* indices. The map vector runs along the *map width*. A simple map would consist of just a single width-height plane; a more complicated map would have a width-height plane for every depth, time, and element plane in the value segment. This provides a great deal of mapping flexibility. For example, every plane in a volume or every image in an animation could have a separate map.

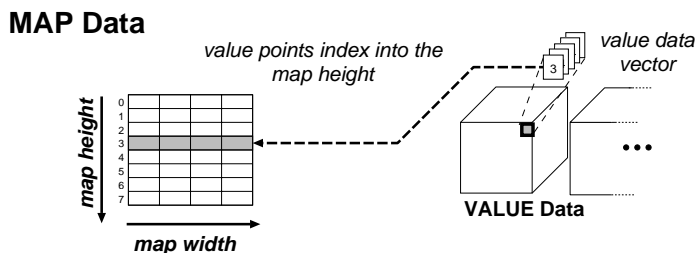
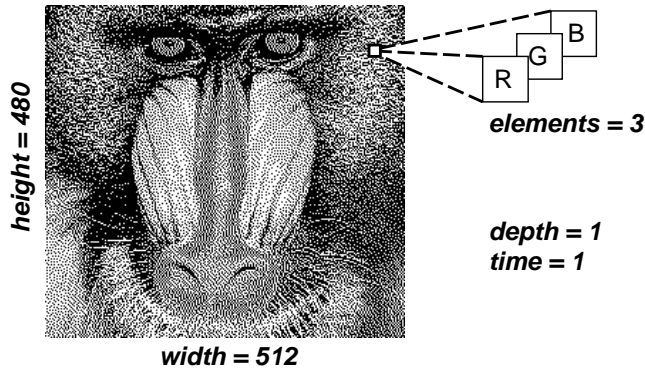


Figure 10: Polymorphic Map Data. The map segment of the polymorphic model is used store a look-up-table of map vectors. Values in the value segment are then used as indices into the map; the value points map to indices along the *map height*. The map vector runs along the *map width*. A number of *map width* x *map height* planes may exist; the map size may match the depth, time, and element size of the value segment by specifying the appropriate *map depth*, *map time* and *map elements*.

D.1.7. Polymorphic Example 1 : Storage of an RGB Image

This example illustrates the storage of a simple RGB image. This image utilizes only the value segment. The image is 512 pixels wide by 480 pixels high. The pixels are stored along *width* and *height* in the value segment. The *depth* and *time* size of the value segment are each 1. The RGB values for each pixel are stored down elements; thus the *element* size is 3. The other segments are not used.

VALUE data

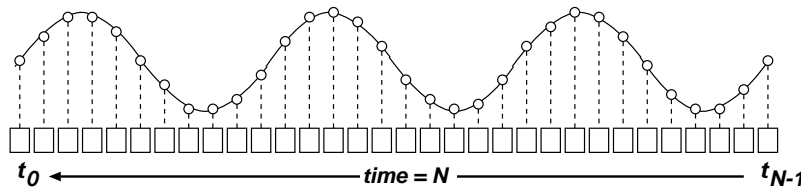


No LOCATION data
 No TIME data
 No MASK data
 No MAP data

D.1.8. Polymorphic Example 2 : Storage of a Signal

This example illustrates the storage of a regularly-sampled time signal. The sampled points are stored in the value segment along time, thus the *time* size is equal to the number of samples N . The *width*, *height*, *depth*, and *element* sizes are all 1. The other segments are not used.

VALUE data



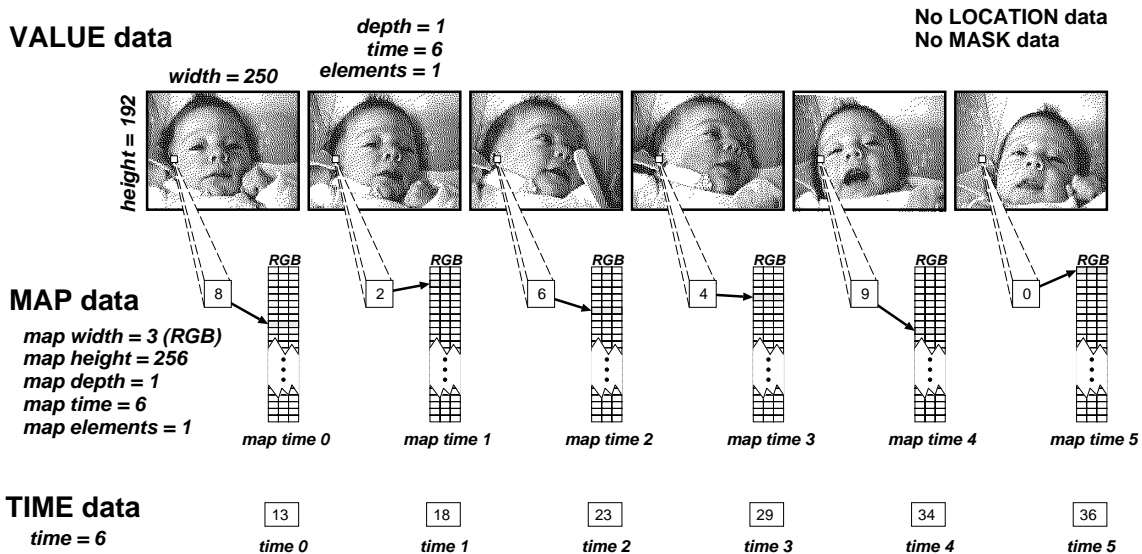
No LOCATION data
 No TIME data
 No MASK data
 No MAP data

width = 1
height = 1
depth = 1
elements = 1

values points represent sampled signal amplitudes
 and are stored down the time dimension

D.1.9. Polymorphic Example 3 : Storage of an Animation with RGB Colormap

This example illustrates the storage of a mapped animation. The frames of animations are stored in six *width* x *height* value planes through *time*. The *depth* and *element* size of the value segment are one. Each point in the value segment maps into the map segment. The values index into the *map height*; there are 256 available RGB vectors in this example. Because the map contains RGB values, the *map width* is three. This map segment contains a single colormap for each frame of animation, thus the *map time* is 6. If the *map time* had been 1, then the entire animation would have referred to a single colormap. The time segment is used to store time-stamps for each frame of the animation. The mask and location segments are not utilized.



D.2. Geometry Data Services

Geometry Data Services is designed to meet the specific needs of geometry and volume storage. The *geometry data model* implemented by this service supports the storage and retrieval of a number of geometric primitives, such as spheres, triangles and lines. Other non-geometric primitives such as octmeshes and textures are also supported. This section of the manual provides a general explanation of the geometry data model, as well as an example of how a geometry data set is stored. You can find specific details about this model in *Chapter 3, Geometry Data Services* of this volume.

D.2.1. Geometry Data Model

The geometry model is centered around a *primitive list*. This list is able to store any combination of geometric primitives such as spheres or polylines. Each geometric primitive consists one or more different *types* of data, such as location data and color data. The types of data required depend on the primitive; all primitives have location and most have color while only some have radii or normals. Most data is explicitly provided, although colors may be provided indirectly via a *colormap*. *Quadmesh* and *octmesh* primitives, which are not illustrated here, are also available. These mesh primitives are overlaid on top of the polymorphic data model. Thus, from the point of view of the polymorphic data model, a quadmesh will appear to be an image, and an octmesh will appear to be a volume.

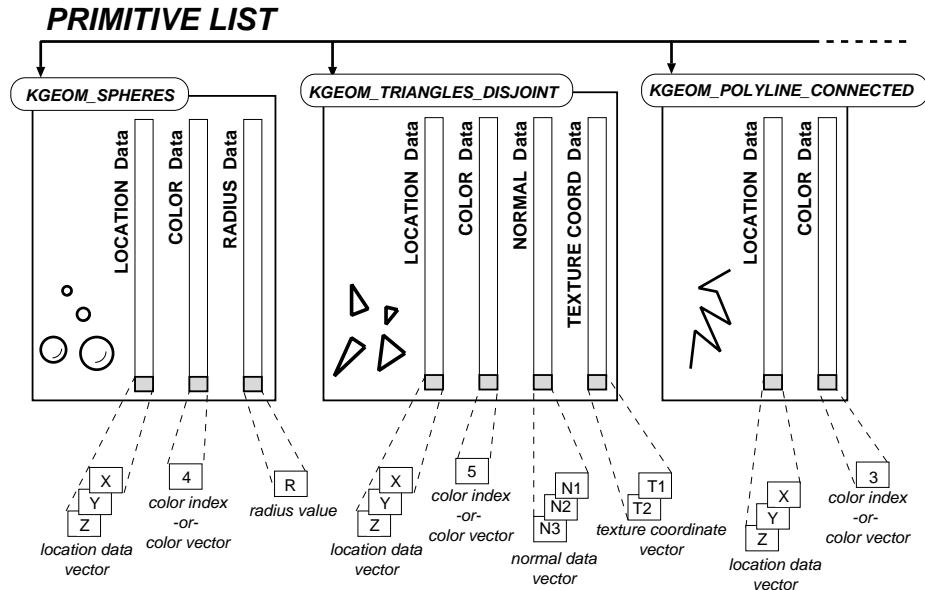
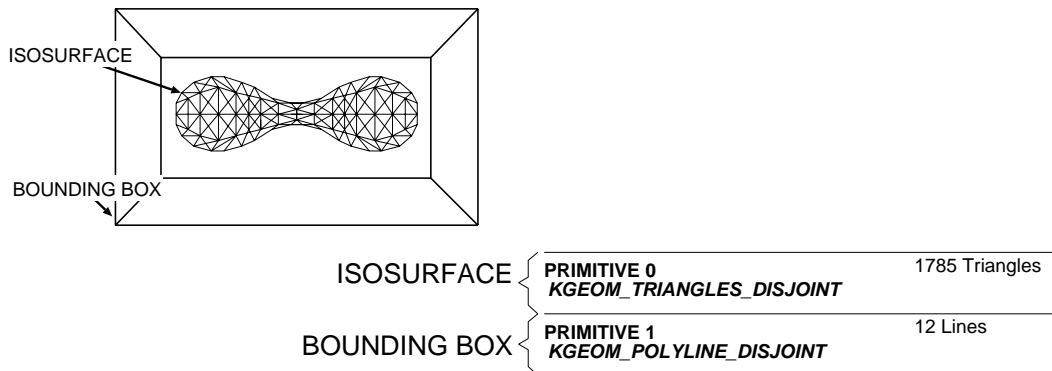


Figure 11: An overview of the Geometry Data Model. The geometry model consists primarily of a *primitive list*. Geometric primitives are stored and retrieved from this list. Each geometric primitive is an aggregate of different types of data. For example, a *spheres* primitive consists of location data, color data and radius data. Please note that a single spheres primitive contains multiple spheres. Most data is explicitly given for each primitive. Color data however may consist either of explicit color vectors or of indices into a colormap. This figure does not illustrate any of the mesh primitives.

D.2.2. Geometry Example: Storage of Geometry Primitives

This example illustrates the storage of an isosurface and a bounding box. The isosurface is constructed from many thousands of disjoint triangles. The triangles composing the isosurface are stored in the first primitive on the list. Please note that the triangles in this one primitive could have been broken into separate primitives if so desired. The bounding box surrounding the isosurface simply consists of 12 disjoint lines. These are stored in the last primitive on the list.



D.3. Color Data Services

Color Data Services provides very specific functionality related to color data. The *color data model* implemented by this service provides a number of automatically generated standard colormaps as well as a number of colormap operations that can be utilized with polymorphic or geometric colormap data. It also provides a number of color interpretation attributes, which indicate how the color vectors in a data object should be interpreted. This section provides a general explanation of the colormap data model. You can find specific details about this model in *Chapter 4, Color Data Services* of this volume.

D.3.1. Color Data Model

The color model provides both autocolor procedures and colormap operations. An autocolor procedure, when invoked, creates a colormap according to the given autocolor scheme. A colormap operation, when invoked, will take an action on the existing colormap. In both cases, the original colormap is saved. The color model also provides a mechanism for storing the current colorspace model of the data and determining whether or not the color vectors contain an alpha channel for storing opacity information.

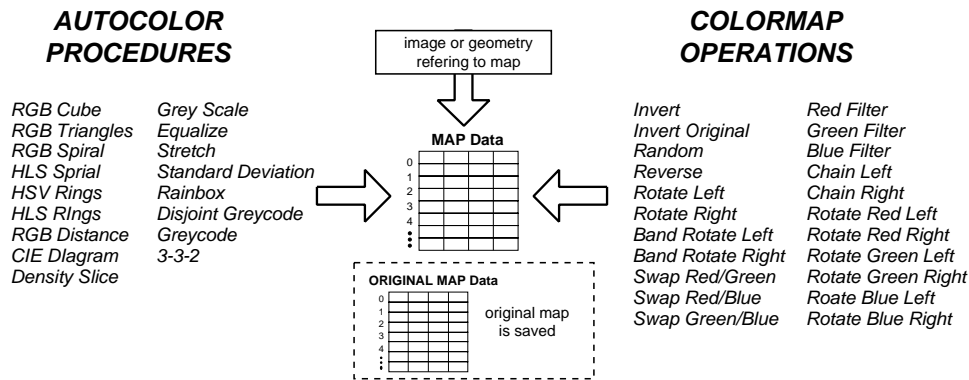


Figure 12: An overview of the Color Data Model. The color model provides both autocolor procedures and colormap operations. Autocolor procedures replace the existing map, while colormap operations operate on the existing map. In both cases, the original colormap is saved. The map can be used by both polymorphic and geometry data.

E. Data Access Presentation

Data Services has the ability to present the data stored within a data object in a variety of different ways. Data can be cast, resized, normalized, or scaled on access. The API to this functionality is provided by a number of data presentation attributes. By setting the appropriate attributes, Data Services returns the data to you in the form that you find the most convenient. In order to understand how the attributes are used, it is necessary to understand how the data object is divided into a presentation layer and a physical layer.

E.1. Presentation and Physical Layers

A data object can be thought of in terms of two layers: a *presentation layer* and a *physical layer*. Attributes at the physical layer typically describe the actual physical characteristics of the data. Attributes at the presentation layer typically dictate how the data is to be accessed. For example, there is a physical data type attribute which indicates what data type the data is actually stored in, and a presentation data type attribute which indicates what data type the data should be presented in. If the presentation data type is set to integer, while the physical data type is set to short, then the data will be cast from short to integer on retrieval and from integer to short on storage.

This presentation capability is handled by the *data pipeline*. This pipeline consists of a number of stages, where each stage is designed to handle a single component of the presentation. Data passes through this

pipeline during `put_data` and `get_data` calls. Only the pipeline stages that are necessary to present the data as requested will be invoked for any given data access. Thus, the data presentation capability can be bypassed if speed is important.

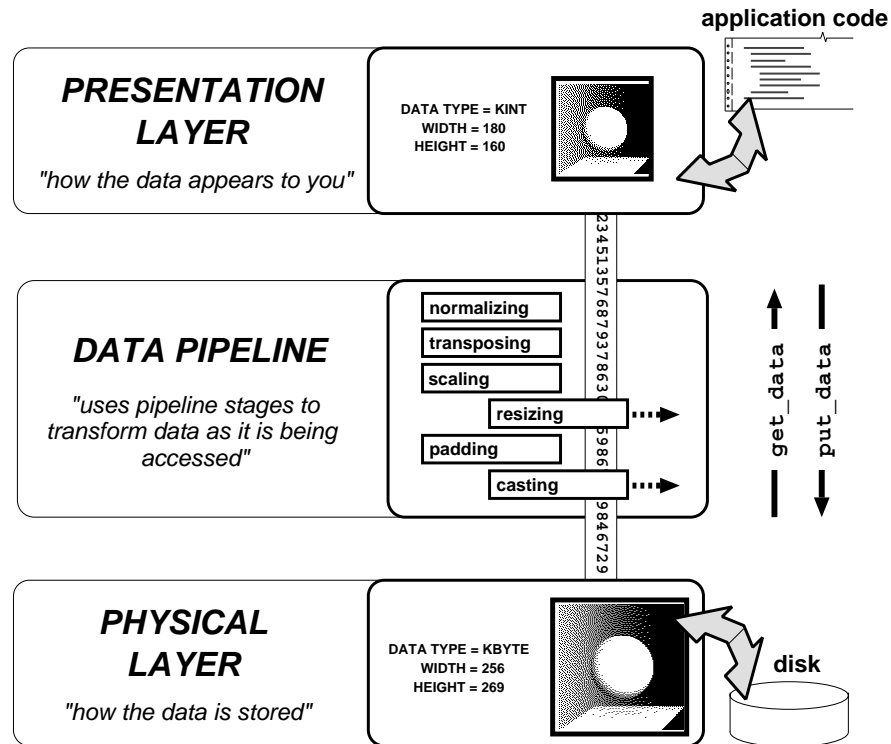


Figure 13: A data object can be thought of as having two layers, a presentation layer and a physical layer. Attributes at the physical layer determine the storage characteristics of the data, such as its size and data type. Attributes at the presentation layer determine the presentation characteristics of the data. On access, data is passed through a data pipeline, which transforms the data according to the presentation attributes. Each presentation attribute corresponds to a stage in the data pipeline; only the necessary stages are invoked on data access.

E.2. Reference Objects

Data Services provides you with the ability to create *reference objects*. A reference object is simply a new presentation layer on an existing data object. A reference object will share the same physical layer as the original object, and as such, will share all the data and the corresponding physical attributes of the original data set. However, the reference object will have its own presentation layer with its own copy of all the presentation attributes. This provides you with the ability to have multiple views of the same data set in a single program. It provides a mechanism by which a single data set can be accessed in two different contexts simultaneously. This powerful concept has a number of uses. Reference objects are most commonly used to limit side effects in data processing libraries, or to provide multiple views of a single data set in an interactive program.

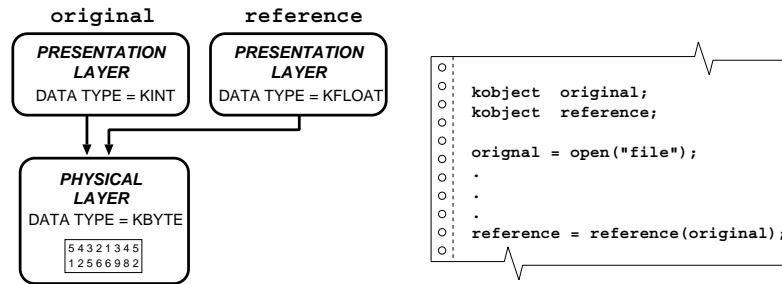


Figure 14: An illustration of Reference Objects. It is sometimes useful to have access to multiple versions of a given data set. By using reference objects, you can avoid having multiple copies of the data and have instead a single physical copy of the data with multiple presentations. Each reference object has its own presentation layer, but they all share a common physical layer. References are made from an original object.

F. File Format Support

Data Services transparently supports numerous file formats. The details of file format support are hidden in the Application Programming Interface (API) in Data Services. When you open up an input data object, the *file formats layer* of Data Services will check the underlying file to determine if it is one of its supported file formats. If it is, the data contained in the file will be made available through a number of data segments. You can then access the data through the various application data models that overlay these segments. With this abstraction, you can open a data object and process it without having to consider its underlying data format.

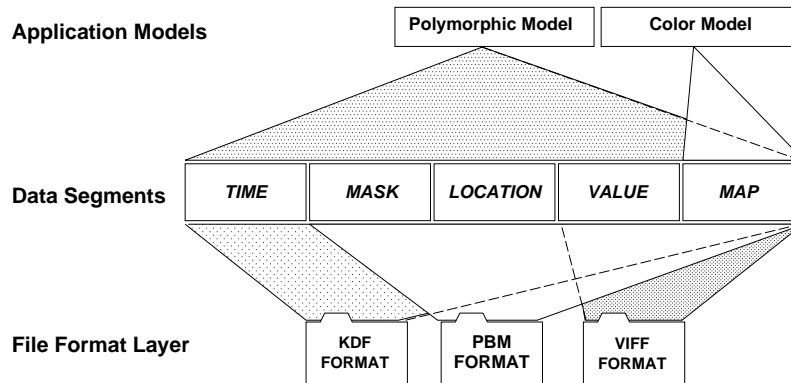


Figure 15: Data Services transparently supports a number of data file formats. The file format layer understands how to read and write several different formats, and is able to present the data to Data Services in the form of different data segments. These data segments are then accessed through the different application data models.

F.1. Supported Formats

Data Services has support for the following file formats.

Data Services Supported Formats	
Name	File Format Description
KDF	VisiQuest Data Format
JPEG	Independent JPEG Group
PNM	Portable Anymap File Formats
PCX	Paintbrush Format
Rast	Sun Raster
Xbm	X Bitmap
Xpm	X Pixmap
Xwd	X Window Dump
Avs	AVS Image Format
Arf	Another Raster Format
Ascii	Ascii formatted data
Raw	Raw data
Eps	Encapsulated Postscript (output only)

F.2. Format Storage Issues

Please note that not all formats are capable of storing all segments. For example, the PBM format is only able to store map and value data. Thus, if you create a data object with explicit location data and then save the object using the PBM format, your location data will be lost. The VisiQuest KDF format is the only supported format which is capable of generally supporting all data segments and attributes. Please also note, however, that since most of the supported formats are designed for storing images, this is typically not a limitation if you are working with image data. In these cases, the file format support provides you with the ability to seamlessly store your data in formats usable by other software systems.

G. Large Data Sets

With other systems, the entire data set is read from disk and placed into memory for processing. This does not work with large data sets, where the amount of data stored on disk exceeds the amount of memory available. Data Services takes a better approach with its treatment of large data sets. If the amount of data in a file exceeds the amount of memory available, then Data Services will read into memory only the data that you specifically request. With Data Services, it is possible to write programs to process large data sets.

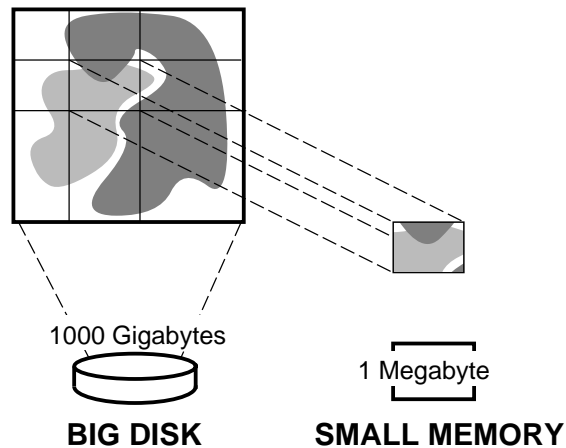


Figure 16: Data Services provides the ability to operate on large data sets. If the entire data set will not fit into memory, then only the data that is requested on any given data access call will be read from the disk. Routines written with data services that never request more data than can be stored in memory will be able to process large sets.

H. Data Services Organization

Data Services consists primarily of two services: *Application Data Services* and *Data Management Services*. Application Data Services encompasses Polymorphic Data Services, Geometry Data Services, and Color Data Services. Application Data Services contains all the public, high-level functionality of Data Services and is typically the only data service you need to be aware of. Data Management Services contains the segment and attribute infrastructure of Data Services, along with the Data Presentation Pipeline. Its API, while publicly available, is intended only for advanced users who wish to bypass the data models imposed by the Application Data Services. Below these two services are the *File Format* libraries. These libraries, which contain no publicly available functions, handles the reading and writing of the different supported data formats.

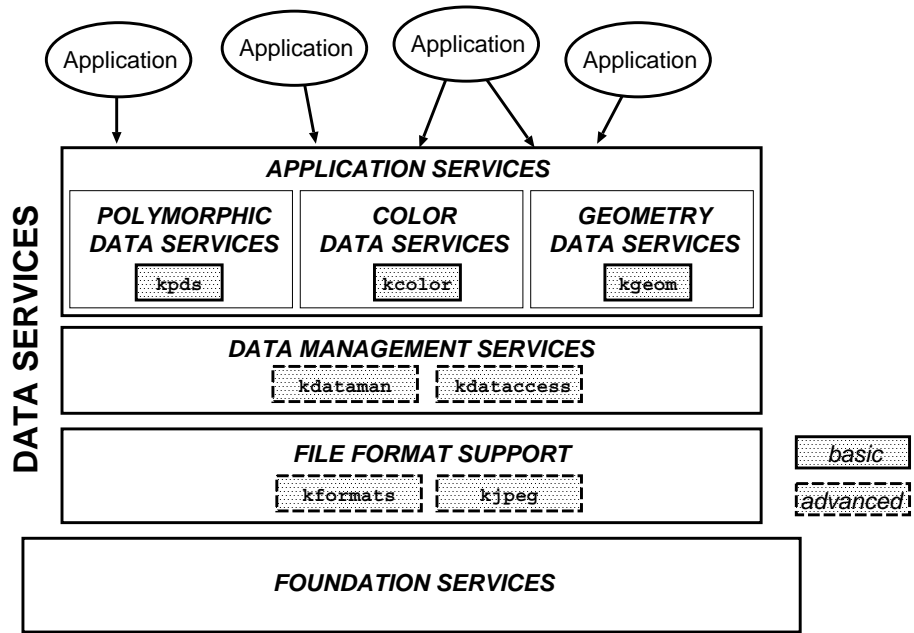


Figure 17: An illustration of the organization of Data Services. Data Services consists of multiple layered libraries. The uppermost layer contains the Application Data Services libraries (kappserv and kgeom). The kappserv library contains Polymorphic Data Services and Color Data Services while the kgeom library contains Geometry Data Services. Below these libraries is the Data Management Services layer. This layer contains the segment and attribute infrastructure of Data Services as well as the Data Presentation Pipeline. This layer is broken into two libraries, kdataman and kdataaccess. The lowest library in Data Services is the File Format library. This library contains readers and writers for all the underlying data formats supported by Data Services. Support for the JPEG format is contained within its own library. Most programmers should only use the application services.

Table of Contents

A. Overview of Program Services	1-1
B. Introduction to Data Services	1-2
C. Application Programming Interface (API)	1-4
D. Overview of the Application Data Services	1-5
D.1. Polymorphic Data Services	1-6
D.1.1. Polymorphic Data Model	1-6
D.1.2. Value Data	1-7
D.1.3. Location Data	1-8
D.1.4. Time Data	1-9
D.1.5. Mask Data	1-9
D.1.6. Map Data	1-10
D.1.7. Polymorphic Example 1 : Storage of an RGB Image	1-10
D.1.8. Polymorphic Example 2 : Storage of a Signal	1-11
D.1.9. Polymorphic Example 3 : Storage of an Animation with RGB Colormap	1-11
D.2. Geometry Data Services	1-12
D.2.1. Geometry Data Model	1-12
D.2.2. Geometry Example: Storage of Geometry Primitives	1-13
D.3. Color Data Services	1-14
D.3.1. Color Data Model	1-15
E. Data Access Presentation	1-15
E.1. Presentation and Physical Layers	1-15
E.2. Reference Objects	1-16
F. File Format Support	1-17
F.1. Supported Formats	1-18
F.2. Format Storage Issues	1-18
G. Large Data Sets	1-19
H. Data Services Organization	1-19

This page left intentionally blank

Chapter 2

Polymorphic Data Services

Chapter 2 - Polymorphic Data Services

A. Introduction

Chapter 1 introduced the concept of an Application-Specific Data Service, which is an Application Programming Interface (API) that is customized for a specific data processing domain or a specific type of interaction with data. Polymorphic Data Services is an Application-Specific Data Service that provides full access to and full utilization of the *Polymorphic Data Model*.

The ability of programs to operate equally well on data from a broad range of application domains is referred to as *polymorphism*. For example, if a program can operate on rasterized image data as well as sampled signals or matrices, then it is polymorphic. The Polymorphic Data Model is a framework for interpreting data that is based on an idealization of the physical universe (please see below). Thus, this model provides a uniform interpretation of data independent of any specific application domain. The model can represent data acquired or generated for applications as diverse as image processing, signal processing, computer vision, numerical analysis, and volume visualization.

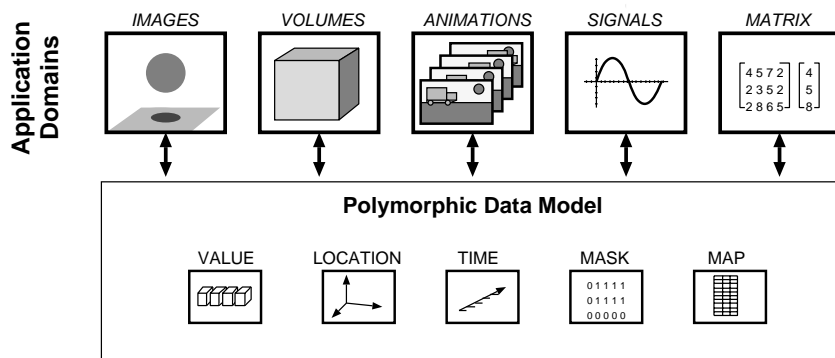


Figure 1: The Polymorphic Data Model is designed to encompass a broad range of scientific application domains. Images, signals, volumes, animations, and matrices can all be stored within this model.

Polymorphic Data Services provides an API that can be easily applied to a wide variety of application domains. However, while the other application services present a model in a form that is more customized for a specific-application domain (such as Geometry Visualization or Color Interpretation), this application service is free of the limitations that may be imposed by such narrow domain-specific interpretations of data.

The remainder of this chapter consists of six sections: (1) an in-depth discussion of the Polymorphic Data Model and (2) how to interact with it, (3) a discussion of the philosophy behind the Polymorphic Data Services API, (4) definitions of the data primitives, (5) definitions of the data attributes and (6) descriptions of the Polymorphic Data Services functions.

B. The Polymorphic Data Model

The *Polymorphic Data Model* is a standardized interpretation that is applied to data that has been sampled from physical phenomena, or that has been artificially generated to emulate physical phenomena. In this model, data is represented as a number of sets, or vectors of information that exist in three-dimensional space and one-dimensional time. Using this model, a *vector* of data can be thought of as consisting of one or more values that exist at a specific location in space-time. Each data point can be considered to be valid or invalid. This collection of properties implies that up to four storage components may be necessary in order to represent the data. For convenience, these "components" are referred to as *segments*.

The four segments that are required to fully represent the model are: *value*, which serves as the primary source of data in the model; *mask*, which provides validity information for each point in the *value*, segment; and *location* and *time*, which together provide explicit world-coordinate placement in three-dimensional space and one-dimensional time. In addition to these four segments, a fifth segment, *map*, is present in the model because it provides exceptional storage compression for quantized data and is a useful representation when visualizing data. These data segments are all interrelated, but are generally accessed independently. These data segments are related together via a set of four indices; w , h , d , t . The aggregation of these data (*value*, *mask*, *map*, *location*, and *time*) allow implicit and explicit data to be dealt with in a convenient manner. Figure 2 illustrates each of the data segments while Figure 3 illustrates the association between the segments with respect to the four indices.

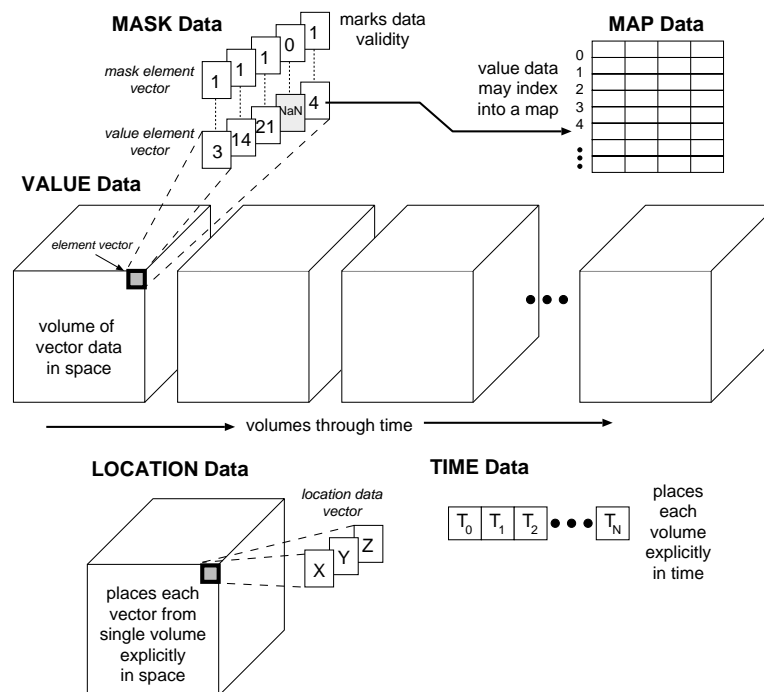


Figure 2: The Polymorphic Data Model consists of five data segments, with each segment serving a specific purpose. The *value* segment consists of data element vectors organized into a time-series of volumes. The volume of value data can be given explicit locations in space with the *location* segment; one location vector is provided for each value vector in a single volume. The volumes of value data can be given explicit locations in time with the *time* segment; a time-stamp may be given for each volume in time. A *mask* segment is available for marking value data validity.

Data is stored or retrieved via simple function calls in units that are referred to as *primitives*. Meta-data, or information which describes the data and helps to provide an interpretation for it (such as its data type or a color space model) are referred to as *attributes* and are similarly manipulated with simple function calls.

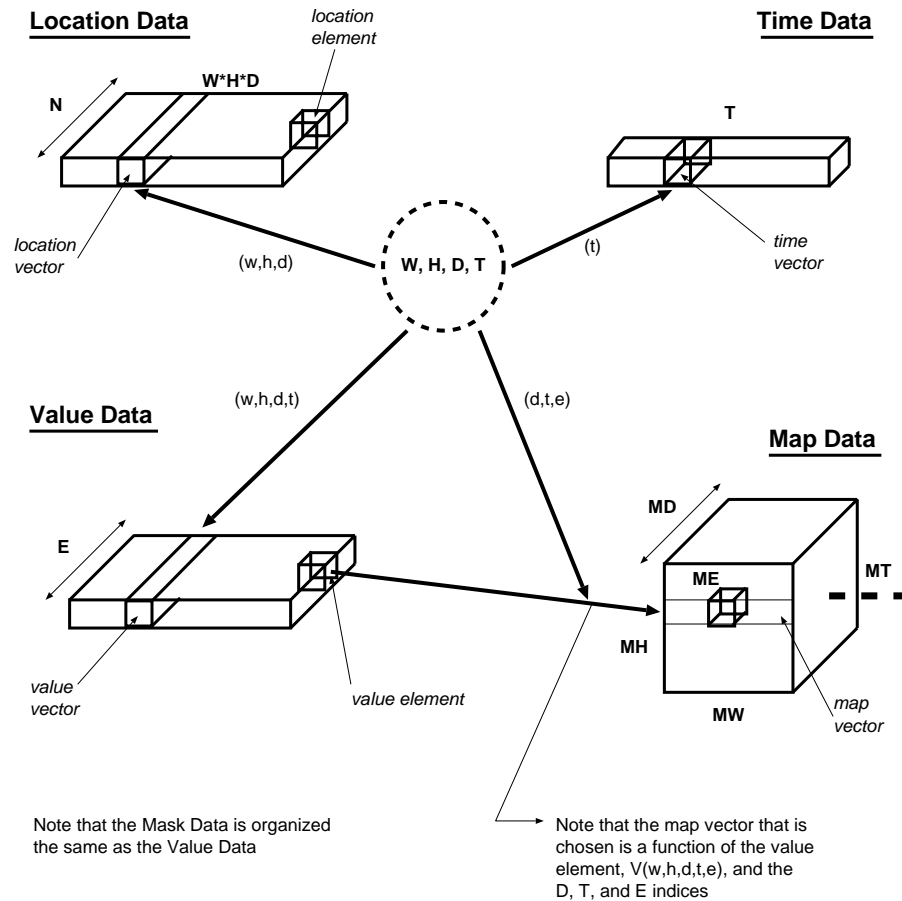


Figure 3: The Polymorphic Data Model is comprised of five data components. The components are related through their indices.

The values in a data vector are acquired from, or reside in, four-dimensional space-time. This is based on the premise that data sets are usually acquired from, or modeled after the physical world. These data vectors exist in one data segment of the model called the *value* data. Each *value* data vector is accessed via four indices, hence it exists in four-dimensional space-time. Along with this *value* data, it is convenient to have associated *mask* data. Within this *mask* data there exist a number of *mask* values, one for every data vector in the *value* data. These mask values are used for marking data validity. When a data set's organization in *location* and *time* is implicit, simplified representations are allowed. For example, an image having no explicitly defined properties is stored as an array of pixels or pixel vectors. Implied relationships between the data sets as well as the implied spatial and temporal locations of the data sets are inherently defined by their arrangement. When these relationships or locations are explicit, a more complicated data representation is needed. This need is satisfied by the *location* and *time* data.

Often, spatially different data vectors will contain identical data elements. If the number of distinct element combinations is relatively small, it makes sense to keep them in a separate list. That way, rather than containing multiple copies of similar data, the *value* data could instead contain simply an index that maps into that separate list. This way, the size of the *value* data can be reduced without reducing the information content. This mapping functionality is provided through the *map* data. When taken together, the *value*, *mask*, *map*, *location* and *time* data form the complete Polymorphic Data Model. The programmer that uses Polymorphic Data Services is given access to all of these defined data segments, and it is actually the decision of the programmer to determine or choose the details of the interpretation of the data beyond the stated relationships of the data segments.

The explicit *location* data can also be accessed according to the values of w , h , and d , although this relationship is not enforced. If the programmer chooses to interpret the *location* indices in this way, then for each unique three-dimensional position in the *value* and *mask* data, there will exist an explicit *location* vector. Similarly, the explicit *time* data can also be accessed according to the value of t . If this interpretation is used, then for each unique one-dimensional time in the *value* and *mask* data, there can exist an explicit *time* value. Thus, it is up to the programmer to provide an interpretation of the *location* and *time* data.

B.1. Value Data

The *value* data segment is the primary storage segment in the Polymorphic Data Model. Most of the data manipulation routines are specifically designed to process the data stored in this segment. The *value* segment is used to hold *value* vectors, which are sets of data of the same size and type, located explicitly or implicitly in time and space. To simplify the following discussion, *value* vectors will be defined as having only one dimension, though they can be N-dimensional. The *value* data can then be represented as *value* vectors in four-dimensional space and time, with each *value* vector having e elements. In this context, three of the dimensions define the spatial location, and one defines the time dimension of a *value* vector. Data values can be then accessed within the *value* vector by indexing into the *elements*.

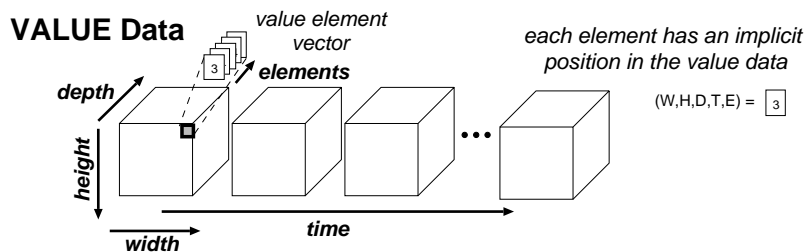


Figure 4: Polymorphic Value Data. The value segment of the Polymorphic Data Model is best pictured as a time-series of volumes. Each volume consists of element vectors oriented implicitly along *width*, *height*, *depth*, *time* and *elements*. Each element can be indexed directly by a 5-tuple position.

When the *location* and *time* of data are implicitly defined, the ordering of the *value* vectors can express relative location and relative time (i.e., which *value* vector is next to which *value* vector). Therefore, *implicit* location and time information is intrinsically contained in the *value* data. If any of these properties is not implied, the explicit data to provide the information can be stored in the *time* and/or *location* data.

B.2. Mask

The *mask* data contain zero and non-zero values located explicitly or implicitly in time and space. Each *mask* data value corresponds in four-dimensional time and space with a *value* vector (if the *value* data is present). Like the *value* data, the *mask* data itself is five-dimensional; three for space, one for time, and one for the *mask* data set, or *mask* vector. The programmer can access points, lines, planes and volumes of *mask* data.

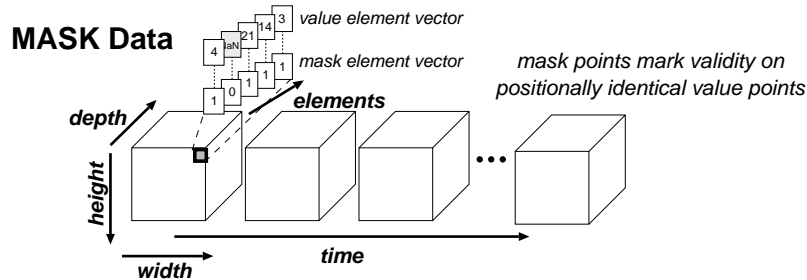


Figure 5: Polymorphic Mask Data. The mask segment of the Polymorphic Data Model is used to mark data validity of the value points. The mask segment is exactly the same size as the value segment.

The programmer can use the *mask* data as a convenient method for indicating the validity of *value* data. The number of values in the *mask* data is equal to the number of values in the *value* data (if the *value* data is present). Like all data segments of the data object, the *mask* data is optional.

All of the *mask* data and *value* data orientation, position and size attributes should be made the same for processing the data object correctly. The dimensions of the *mask* and *value* data must be the same. It is not enforced while actually processing the data object, i.e. it is possible to have different sizes for the *value* and *mask* data, but before the data object is closed the dimensions should be the same. When accessing the *mask* data and the *value*, it is important to have the orientation and position the same for each so the accessed data vectors correspond.

B.3. Map

The *value* data may optionally represent references to data that are held in the *map* data. In this case, each value in the *value* data acts as a pointer or index to a *map* vector located in the *map* data. The *map* data is five-dimensional; *map width* (m_w) is the dimension on which the *map* vector is defined. The other four dimensions are defined as the *map height* (m_h), *map elements* (m_e), *map depth* (m_d), and *map time* (m_t) dimensions. The *map height* corresponds with the number of discrete values between the maximum and minimum values allowed in the *map* data. Note that this *map height* is limited by the *value* data's data storage type. For example, byte storage in the *value* data corresponds with a maximum range of 256. Note also that it is not possible to have maps associated with floating-point data at this time.

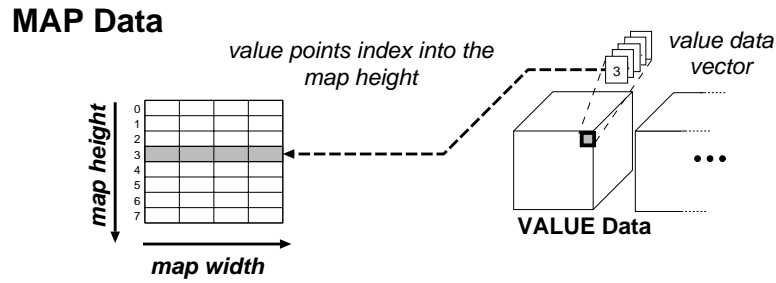


Figure 6: Polymorphic Map Data. The map segment of the Polymorphic Data Model is used store a lookup table of map vectors. Values in the value segment are then used as indices into the map; the value points map to indices along the *map height*. The map vector runs along the *map width*. A number of *map width* x *map height* planes may exist; the map size may match the depth, time and element size of the value segment by specifying the appropriate *map depth*, *map time*, and *map elements*.

A *map* vector is similar to a *value* vector in the *value* data except that it is accessed using a different set of indices. The programmer can access *vectors* of *map* values using *mh*, *me*, *md* and *mt* to specify the vector.

Figure 1 shows how a point of *value* data and its position are used to index into the *map* data. The value of the point is typically used as *mh*, but is not tied to *mh*, *me* is tied to *e* from the *value* or *mask* data or set to 1, *ms* is set to 1 or tied to either the *w*, *h*, or *d* common index, and *mt* is tied to *t* common index or set to 1.

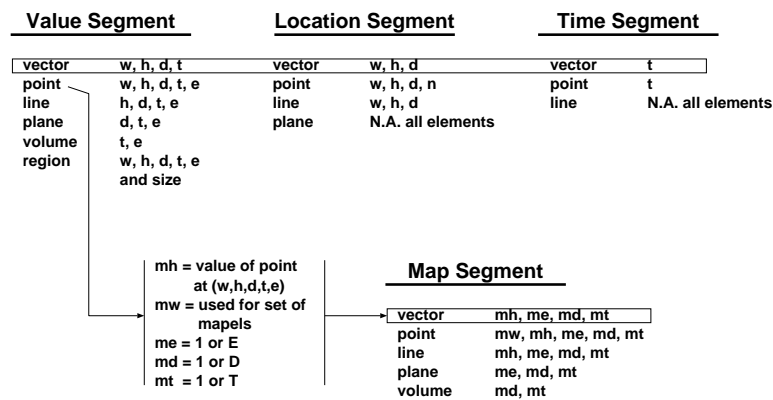


Figure 7: Another view of the Polymorphic Data Model and the interrelationships between each data type. The diagram also lists the indices and primitives of each data segment.

B.4. Location

When the relative location implied by the accessing order of the *value* data is insufficient for defining location information, explicit *location* data is required. When using explicit *location* data, each *value* vector in the *value* data has a corresponding *location* vector in the *location* data. Each collection of *value* vectors has a specific time, but a *value* vector's explicit spatial location can NOT change with time. In other words, when

the *time* dimension is greater than one, the same *location* data will be applied for all time.

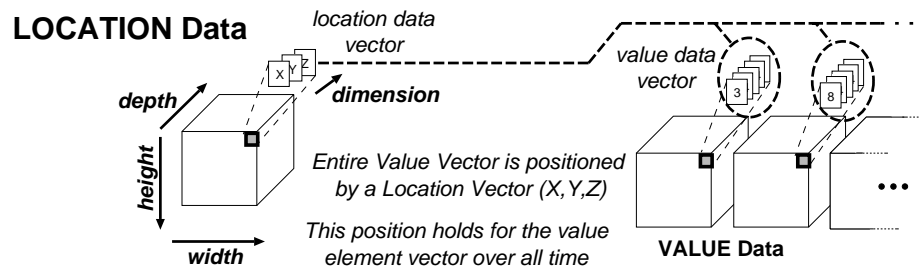


Figure 8: Polymorphic Location Data. The location segment of the Polymorphic Data Model is used to explicitly position the volume vectors in space. The location segment consists of a volume of location vectors; the *width*, *height*, and *depth* of this volume is shared from the value segment. The location vector is of size *dimension*.

The *location* data is made up of *location* vectors, one per *value* vector. The number of *location* vectors is equal to the product of *width* * *height* * *depth*. The size or dimensionality (*n*) of a *location* vector is assigned using the size attribute of the *location* data. A specific *point* of a *location* data is found using *w*, *h*, *d*, *n*, while a specific *location* vector is found using the indices *w*, *h* and *d*. For three-dimensional *location* data, *n* would be 3. Thus a *location* vector is tied to a *value* vector via the *w*, *h* and *d* common indices illustrated at the center of Figure 2.

B.5. Time

When the relative time implied by the accessing order of the *value* data is insufficient, explicit *time* data is required. Such a circumstance can occur when the value data is irregularly sampled in time. The *time* vector is indexed using *t* and is a single element of data specifying explicit time of a *value* vector located at *w*, *h* and *d*.

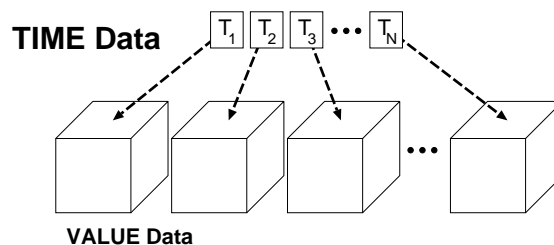


Figure 9: Polymorphic Time Data. The time segment of the Polymorphic Data Model is used to explicitly position the value volumes in time. The location segment consists of a linear array of time stamps; the number of timestamps is equivalent to the *time* size of the value segment.

C. Interaction with the Polymorphic Data Model

Polymorphic Data Services provides a set of standard units of access for data that is held in a `kobject`. Throughout this chapter, these units are referred to as *primitives*. The basic primitives that are available are: *points*, *lines*, *planes*, and *volumes*. In addition to these, Polymorphic Data Services defines three other specialized primitives: *vectors*, *regions*, and *all*.

The properties of the data accessed via the function calls listed in Section G, *Functions Provided By Polymorphic Data Services* of this chapter are controlled by the attributes of the data presented in Section F, *Attributes Defined by the Polymorphic Data Model* of this chapter. It may be tempting for the reader to associate terms like *signal* and *image* to the terms *line* and *planes*. The reader may also desire examples in the context of signals and images. However, the generalized Data Object implemented as the `kobject` is not defined in terms of images and signals, since these concepts are too specific. Sub-classed data models can be defined and built from the more general PDS being presented here to create programming models for application-specific areas as discussed in earlier chapters.

C.1. Presentation of the Data Object

Polymorphic Data Services has the ability to present the data stored within a data object in a variety of ways. Data can be cast, resized, normalized, scaled, or re-oriented on access. The API to this functionality is provided by a number of attributes. By setting the appropriate attributes, Polymorphic Data Services will return the data in the form that is most convenient to process. In order to understand how the presentation attributes are used, it is necessary to understand how the data object is divided into a presentation layer and physical layer.

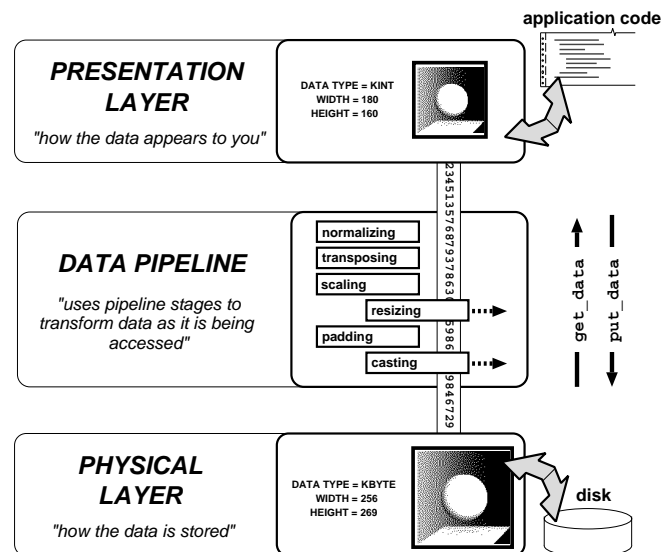


Figure 10: A data object can be thought of as having two layers, a presentation layer and a physical layer. Attributes at the physical layer determine the storage characteristics of the data, such as its size and data type. Attributes at the presentation layer determine the presentation characteristics of the data. On access, data is passed through a data pipeline which transforms the data according to the presentation attributes. Each presentation attribute corresponds to a stage in the data pipeline; only the necessary stages are invoked on data access.

A data object can be thought of in terms of two layers: a *presentation layer* and a *physical layer*. Attributes at the physical layer typically describe the actual stored characteristics of the data. Attributes at the presentation layer typically dictate how the data is to be accessed. For example, there is a physical data type attribute that indicates in which data type the data is actually stored and a presentation data type attribute that indicates in which data type the data should be presented. If the presentation data type is set to integer while the physical data type is set to short, then the data will be cast from short to integer on retrieval and from integer to short on storage.

The following sections outline the different mechanisms that are available for customizing data access.

C.2. Casting

The *casting* feature provided by Polymorphic Data Services is used to change the data type from the type stored to another data type that is more suitable for processing. This process is nearly automatic. What is involved is setting the data type of the segment that is being operated on to the desired processing data type with a call to `kpds_set_attribute()` or `kpds_set_attributes()`. Afterward, all data retrieved with `kpds_get_data()` will be returned to the user in the data type specified, regardless of the stored data type. If operating on an output object, then setting the data type of the output object to something different from the data type being stored informs Polymorphic Data Services that any data being written via a call to `kpds_put_data()` will be given in the specified data type, but should be cast before being written out.

The casting feature is performed via the ANSI C cast operation. Since ANSI C does not dictate the behavior of certain lossy cast operations such as signed information cast to an unsigned data type, the behavior of this operation in certain circumstances can be different from platform to platform.

C.3. Scaling and Normalization

Scaling and normalization are two activities that alter the range of data when presented to the user. These operations are often necessary when processing data where an algorithm operates better on a limited range of data. After indicating that scaling or normalization is to occur, any call to `kpds_get_data()` will cause the range of the data to be altered before it returned to the calling program. The attribute `KPDS_VALUE_SCALING` (and similar attributes for the other polymorphic segments) determines what kind of range alteration is to occur. The default value for this attribute is `KNONE`, which indicates that no scaling whatsoever is to occur. Other legal values for this attribute are `KSCALE` and `KNORMALIZE`.

When the scaling attribute is set to `KSCALE`, then the range of the data is controlled by two attributes: `KPDS_VALUE_SCALE_FACTOR` and `KPDS_VALUE_SCALE_OFFSET`. The range change is computed by applying the scale factor first to each data point, then adding the scale offset.

When the scaling attribute is set to `KNORMALIZE`, the range of the data is controlled by two other attributes: `KPDS_VALUE_NORM_MIN` and `KPDS_VALUE_NORM_MAX`. These two attributes indicate the minimum and maximum magnitude of the data. The effective scale factor and offset are computed by examining every point in the primitive that was accessed via `kpds_get_data()` or `kpds_put_data()`. Thus, this is not a global normalization over the entire set, but rather a local normalization over the extent of the data being accessed.

It is important to note the order in which all of these presentation changes are applied. The normalize and scale operations occur after the cast operation if the cast operation is converting to a "higher order" data type, i.e. a data type that has a higher range or precision. If casting from a higher order data type to a lower order data type, i.e., one that has less range or precision, then the Normalization or scaling occurs before the cast

operation.

C.4. Padding and Interpolation

Padding and interpolation are operations that change the apparent size of the data set being accessed. These operations are useful in circumstances in which a particular size of data is required in order for an algorithm to function properly, such as a Fast Fourier Transform or in instances in which two operands must be the same size in order for the algorithm to behave in a predictable manner, such as an addition operation. Other instances where interpolation is useful is in visual applications for zooming or panning windows. This behavior is controlled by an attribute called `KPDS_VALUE_INTERPOLATE`. This attribute can be set to one of three values: `KNONE`, `KPAD`, or `KINTERPOLATE`. The default value of this attribute is `KPAD`. When this attribute is set to `KNONE`, it indicates that access of data outside of the physical bounds of the data set should not be permitted. If a program attempts to access data that lies beyond the bounds of the data set in this mode, Polymorphic Data Services will generate an error.

If set to use the `KPAD` mode, Polymorphic Data Services will allow access of data outside of the physical bounds of the data set. Any data that is retrieved that is not part of the data set will be set to a constant value indicated by the `KPDS_VALUE_PAD_VALUE` attribute. This attribute takes two `double` arguments that represent the real and imaginary component. The imaginary component is only used if the data type being returned is complex.

If the presentation size is set to be larger than the physical size, then any data that falls outside of the bounds of the data set will similarly be set to this pad value. This mode also allows the presentation size to be set to a value that is smaller than the physical size. In this mode, data outside of the presentation size is clipped, i.e., it is not accessible.

If the `KPDS_VALUE_INTERPOLATE` attribute is set to `KZERO_ORDER`, then this indicates that the difference in the presentation size and the physical size of the data segment should be rectified via a zero-order-hold (i.e., pixel replication) interpolation. Currently this is the only true interpolation mode available in Data Services. If the presentation size is larger than the physical size, then an adjacent data point is replicated for each point that does not exist in the interpolated data set. If the presentation size is smaller than the physical size, then the data set is sub-sampled to produce a smaller version of the original data.

C.5. Conversion of Complex Data

Complex conversion can be thought of as an extension to casting. However, since the process of converting data from a complex data type to a non-complex data type, or vice-versa, is uniquely lossy, this capability is provided as a separate feature so that its behavior can be more easily controlled.

This control is provided via the `KPDS_VALUE_COMPLEX_CONVERT` attribute (and its sister attributes for each of the other polymorphic segments). This attribute determines how to translate real valued data into complex data. For example, if the `KPDS_VALUE_COMPLEX_CONVERT` attribute is set to `AccuSoftEAL`, the real valued data will be interpreted as the real part of the complex pair. Similarly, a setting of `KIMAGINARY` instructs Data Services to interpret the data as the imaginary component. In either case, the other component of the pair is set to zero. When `KPDS_VALUE_COMPLEX_CONVERT` is set to `KMAGNITUDE`, then the magnitude of the complex pair is set to the value of the data. Currently, this is performed by setting the phase to 0 radians. Thus, `KMAGNITUDE` has the same effect as `AccuSoftEAL`. If the `KPDS_VALUE_COMPLEX_CONVERT` attribute is set to `KPHASE`, then the real valued data is interpreted as radian data and the magnitude is set to

1.0.

When complex data is returned to the application from Data Services, it will be in the form of a `kcomplex` or `kdcomplex`. There is a complete set of operator functions available for operating on these data types. These functions are available in the `kmath` library. When operating on complex data, the application programmer is encouraged to refer to the *kmath library for information on complex operations*.

C.6. Map Evaluation

The *map* segment is typically used as a means to reduce the overall space of a data set. It contains vectors of information that are referenced by indices in the *value* segment. This representation is ideal for some applications such as color manipulation or visualization, but this representation is not usually convenient for data processing operators. Polymorphic Data Services provides a means of mapping the indices in the *value* segment through the *map* segment "on the fly" during data access with `kpds_get_data()`. The affect is that size of the `KELEMENTS` axis of the *value* segment is multiplied by the size of the `MAP_WIDTH` axis of the *map* segment. The `KELEMENTS` axis will now contain true data instead of indices into the map data.

This behavior, by default, is disabled. To enable automatic mapping of data, the `KPDS_MAP_ENABLE` attribute should be set to `KMAPPED`. If no *map* data exists, then this attribute is ignored.

This functionality, while powerful in the sense that it can drastically simplify writing a fully polymorphic operator, has a significant overhead. This is because mapping data is inherently a random access process, and Polymorphic Data Services must provide a fully general solution. If performance is critical to an application, it is advisable to perform this operation manually, because typical applications can make simplifying assumptions about the nature of the data and the maps that cannot be made by Polymorphic Data Services.

C.7. Mask Evaluation

Another operation provided by Polymorphic Data Services that simplifies writing polymorphic operators is dynamic evaluation of *mask* information. By default, the *mask* segment is ignored, and the programmer must write applications to interpret the *mask* manually. By setting the `KPDS_MASKED_VALUE_PRESENTATION` to `KUSE_SUBSTITUTE_VALUE`, Polymorphic Data Services will replace every data point in the *value* segment that has a corresponding 0 value in the *mask* segment to the value specified in the `KPDS_MASK_SUBSTITUTE_VALUE` attribute. This functionality is useful when there is an "identity" value that can be used in place of invalid data without adversely affecting the results of a computation. For example, in visualization, it might be reasonable to replace all invalid data with a pixel of a particular color. Using this functionality, this capability is trivial to implement.

Similar to map evaluation, this functionality adds overhead to the retrieval of data. However, since the *mask* segment is accessed the same way that the *value* segment is accessed, the overhead is not as severe.

C.8. Axis Assignment

The polymorphic data model is organized around an axis system that defines three spatial dimensions: *width*, *height*, and *depth*. Polymorphic Data Services provides an attribute called `KPDS_AXIS_ASSIGNMENT` that allows these three spatial axes to be reordered or reassigned. This means that a program can change the meaning of the *width* axis to mean the *height* axis. This is useful for applications such as a matrix transposition operation. By simply swapping the *width* with the *height*, a transpose operation is affected at the time of data

access. This operation does not alter data, but rather alters its interpretation and the way in which it is accessed.

C.9. Data Ranging

When interpreting a dataset, it is often useful to understand what the possible range of the data is. For instance, when displaying a dataset as a grayscale image, the minimum data value can be mapped to black and the maximum value can be mapped to white. However, there typically is no way to determine what range a data set occupies without examining each data point. Even then, the actual available data may not occupy the entire expected range. The `KPDS_DATA_RANGE` and `KPDS_DATA_FORMAT` attributes provides a method for storing the range.

The `KPDS_DATA_RANGE` attribute allows data objects to carry along a theoretical range. This attribute is not automatically created for an object. If it does not exist, then there is no expected range.

The `KPDS_DATA_FORMAT` attribute provides a front end for `KPDS_DATA_RANGE` that defines many common predefined data ranges. A program can check this attribute first to see if the `KPDS_DATA_RANGE` attribute should be examined. Predefined data ranges are:

```
KNONE:          raw data
KUBYTE:         0 - 255
KUSHORT:        0 - 65535
KFLOAT:         0 - 1
KUSERDEFINED:   User-defined range; must have been set
                using KPDS_DATA_RANGE.
```

The routine `kapu_scale_data` sets the `KPDS_VALUE_SCALE_OFFSET` and `KPDS_VALUE_SCALE_FACTOR` attributes so that data accessed will be scaled to the indicated range. Scaling will be applied to either the value or the map segment as appropriate.

The routine `kapu_minmax` finds the minimum and maximum values for a data primitive.

A usage example is shown below.

```
kpds_get_attribute(src_obj, KPDS_DATA_FORMAT, &data_format);

if (data_format != KFLOAT)
    kapu_scale_data(src_obj, 0.0, 1.0);

kpds_set_attribute(dst_obj, KPDS_DATA_FORMAT, KFLOAT);
```

Please note that these attributes are newly implemented and not widely used through the existing VisiQuest system.

C.10. Reference Objects

Polymorphic Data Services provides you with the ability to create *reference objects*. A reference object is simply a new presentation layer on an existing data object. A reference object will share the same physical layer as the original object, and thus will share all the data and the corresponding physical attributes of the original data set. However the reference object will have its own presentation layer with its own copy of all the

presentation attributes. This provides you with the ability to have multiple views of the same data set in a single program. It provides a mechanism by which a single data set can be accessed in two different contexts simultaneously. This powerful concept has a number of uses. Reference Objects are most commonly used to limit side effects in data processing libraries, or to provide multiple views of a single data set in an interactive program. Figure 11 illustrates this concept.

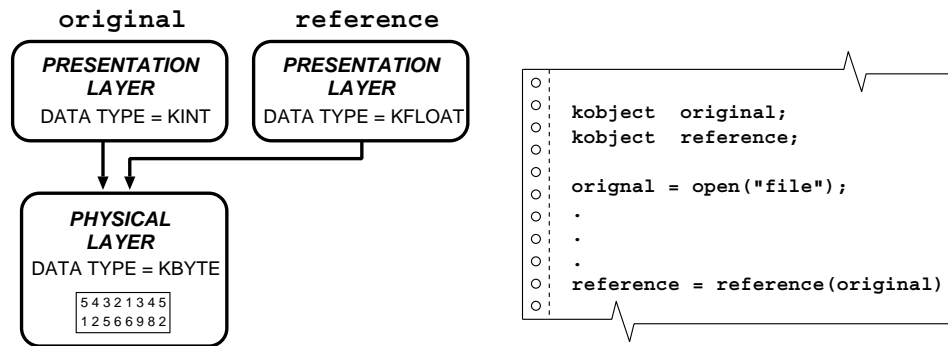


Figure 11: An illustration of Reference Objects. It is sometimes useful to have access to multiple versions of a given data set. By using reference objects, you can avoid having multiple copies of the data and instead have a single physical copy of the data with multiple presentations. Each reference object has its own presentation layer, but they all share a common physical layer. References are made from an original object.

C.11. Auto Incrementing

The *auto-increment feature* provided by Data Services maintains corresponding positions in most of the segments in an abstract data object. For example, if a plane of data is obtained from the value segment of an object then the position in the value data segment is incremented prior to the next read so that the next plane will be obtained. At the same time, any segments with indices tied to the value segment's indices are updated accordingly. For example, the width, height, and depth indices in the location segment will be incremented so that a read from the location segment will result in location data that corresponds to the value data already obtained.

An increment and update operation is performed prior to any second consecutive read or write operation on a segment. This ensures that data can be obtained from every segment necessary before an Increment operation is performed.

State information is maintained internally for each segment to help to determine when it is appropriate to perform an increment operation. Whenever the position attributes (KPDS_VALUE_POSITION, KPDS_MASK_POSITION, KPDS_MAP_POSITION, KPDS_LOCATION_POSITION, KPDS_TIME_POSITION) are set using `kpds_set_attribute` or `kpds_set_attributes` the segment whose position is being set will reset the state information. This means that after setting the position, the next read or write operation will not perform a pre-increment.

D. The Application Programming Interface (API)

This section presents an overview of the Polymorphic Data Services Application Programming Interface (API). It will demonstrate how an object is instantiated, how its attributes are manipulated, how the data is processed and how the object is closed.

An application that manipulates data using the Polymorphic Data Services will be processing data that can be used by other application services using the Polymorphic Data Model.

1. The first step is to instantiate the source and destination data objects. The source object is opened using `kpds_open_input_object()` (for read only). As such, all changes done to the source object do not effect the permanent transport. The destination object is opened with `kpds_open_output_object()` (for write only). All changes to the destination object will be stored in the permanent transport.

```
/*
 * source object
 */
source_object      = kpds_open_input_object("name");

/*
 * destination object
 */
destination_object = kpds_open_output_object("name2");
```

2. This example processes only the *value* data, but the object may contain *map*, *mask*, *time*, or *location* data and data/object attributes. So that data and attribute settings are not lost, the destination object will be made into a copy of the source object. The `kpds_copy_object()` function copies all object data and attributes from the source object to the destination object, so it is not necessary to copy the data and attributes individually. By doing this, it is possible to avoid the side effect of losing data and attributes that could be used by other routines later in some series of processing steps.

```
/*
 * copy object
 */
kpds_copy_object(source_object, destination_object);
```

3. The data will be processed using vector units, where a vector is defined to span the element dimension of the value segment. The attribute `KPDS_VALUE_VECTOR_INFO` will tell us the number of data elements contained in each vector, as well as the number of vectors contained in the entire data set,

```
kpds_get_attribute(source_object, KPDS_VALUE_VECTOR_INFO,
                  &vector_size, &num_vectors);
```

4. The next step is to set the values of the attributes of the Polymorphic Data Model and its primitives so that the data is properly presented, managed and accessed. The following invocations of `kpds_set_attribute()` will cause the source data to be converted to double, and when data is put to the destination object via `kpds_put_data()`, it will be converted to float. As stated

above in step 1, since the source object was opened with `kpds_open_input_object()`, the actual data in the permanent transport is not changed to double. It is converted to double by data services when `kpds_get_data()` is called. However, the destination object was opened with `kpds_open_output_object()`, so the data in the permanent transport will be stored as float when the `kpds_put_data()` is called. By setting the `KPDS_VALUE_DATA_TYPE` on the source object, we can write our `process_data()` so that it only needs to process double data and does not have to handle multiple data types.

```
kpds_set_attribute(source_object, KPDS_VALUE_DATA_TYPE, KDOUBLE);
kpds_set_attribute(destination_object, KPDS_VALUE_DATA_TYPE, KFLOAT);
```

5. Now that the source and destination objects have been set up, instantiated and the attributes appropriately manipulated, the data can be processed. The data object may contain one vector, but most likely it will have a series of vectors as defined by the Polymorphic Data Model. Therefore, the for loop will process each vector. It is important to note that a vector may not be very large so a data primitive that obtains larger parts of the data may be desired. See Tables 5 for more primitives that can be used to access the *value* data. A vector is a good size to use if it is important to process large data sets.

The `kpds_get_data()` and `kpds_put_data()` functions are auto-incrementing. These functions automatically increment the position attribute of the primitive being accessed so that the next execution of `kpds_get_data()` and `kpds_put_data()` access data at the next consecutive position, i.e., the next vector. Therefore, it is not necessary to set the position for the two objects using the `KPDS_VALUE_POSITION` attribute. The default for the position is 0 for all dimensions (*width*, *height*, etc.) when the object is instantiated, so we do not have to set the initial position either. See Tables 6 through 10 for attribute defaults.

```
double *vector = NULL;
for (i=0, i<num_vectors, i++) {
    vector = (double *) kpds_get_data(reference_object, KPDS_VALUE_VECTOR,
                                     (kaddr) vector);
    /*
     * your application function
     */
    process_data(vector, vector_size);
    kpds_put_data(destination_object, KPDS_VALUE_VECTOR, (kaddr) vector);
}
```

The vector variable is set to `NULL` initially so that the function `kpds_get_data()` will allocate the memory for the initial vector of data. Every call after that the call to that vector will have a valid address so that `kpds_get_data()` will just replace the data contained in vector with the new vector and not allocate more memory. See Section G, "Functions Provided By Polymorphic Data Services" of this chapter for more information on the behavior of `kpds_get_data()`.

6. Finally, the data objects should be closed with the Polymorphic Data Services `kpds_close_object()` function. This will store the destination object in the permanent transport and free up memory used by Data Services for the objects.

```
kpds_close_object(destination_object);
kpds_close_object(source_object);
```

This next example will show the steps needed when creating a data part in a data object. It will instantiate an object, create a data part, manipulate some attributes, generate data and then close the object.

1. The first step is to instantiate the destination data object as we did in step 1 of the above example using `kpds_open_output_object()`.

```
/*
 * destination object
 */
destination_object = kpds_open_output_object("name");
```

2. To create the *value* data part, the routine `kpds_create_value()` is used. It will create *value* data that obeys the Polymorphic Data Model.

```
/*
 * create value data
 */
kpds_create_value(destination_object);
```

3. The `kpds_create_value()` does not set the **size** of the data or the **data type**, so they must be set before any `kpds_get_data()` or `kpds_get_data()` calls are done. Setting the **size** and **data type** must be done after any `kpds_create_xxx()` calls. To set the size and data type, `kpds_set_attributes()` is used. In this case, the **data type** will be double and the value data will be 200 x 200 x 1 x 1 x 3.

```
kpds_set_attributes(source_object,
                    KPDS_VALUE_DATA_TYPE, KDOUBLE,
                    KPDS_VALUE_SIZE,      200, 200, 1, 1, 3,
                    NULL);
```

Now we have specified the minimum amount of information about the *value* data, *size* and *data type*, so that Polymorphic Data Services can manipulate it.

4. Now to generate the data, in this case the data primitive `KPDS_VALUE_VECTOR` is used. We declare an array the size that matches the elements of the data, 3. The product of the width and the height give us the number of element vectors to store in the value data, 200 x 200. Putting the data into the destination object with the routine `kpds_put_data()`.

```
double element_vector[3];
for (i=0, i < 200 * 200, i++) {
    /*
     * call a function to generate an element vector for each element of data.
     */
    generate_data(element_vector);
    kpds_put_data(destination_object, KPDS_VALUE_VECTOR,
                  (kaddr)element_vector);
}
```

5. Finally, the data object should be closed with a call to the `kpds_close_object()` function. This will close the source object for reading, write out/save the destination object, and free up any memory that was used by Data Services.

```
kpds_close_object(destination_object);
```

E. Polymorphic Primitives

Data is retrieved and stored in data objects via *primitives*. Primitives are data units that have been defined to allow easy access of the associated data. Two functions have been provided for accessing primitives of data, `kpds_get_data()` and `kpds_put_data()`. These functions take three arguments. The first argument is the object associated with the data, the second argument is the primitive that is desired and the third argument is a pointer to a data buffer that is to be written to or read from.

The `kpds_get_data()` and `kpds_put_data()` functions auto-increment the position indices, or attributes, of the primitive being addressed. The indices column in the following tables lists the indices that are used to set or get individual primitive positions. The order of the indices in the table specifies how the data is organized in the Polymorphic Data Model.

The `kpds_get_data` call will return a type `kaddr`. Note that the `kaddr` type is a generic pointer that is intended to be cast to the proper built-in type or data structure. For example, the data returned should be cast to the proper type as in Step 5 of Section D, "The Application Programming Interface," in this chapter.

E.1. Value Primitives

Table 1 lists the primitives defined by the polymorphic model for *value data*. These primitives are stored in and retrieved from the value data using the `kpds_put_data()` and `kpds_get_data()` functions respectively.

Table 1 - Value Primitives			
Primitive	Dimension	Default Indexing	Description
KPDS_VALUE_POINT	0D	w, h, d, t, e	A pointer to a single <i>value data point</i> . The location of the data point is specified by the <code>KPDS_VALUE_POSITION</code> attribute. After every get/put pair, or consecutive get or put operations, the position attribute <code>KPDS_VALUE_POSITION</code> will be automatically incremented to the next point.
KPDS_VALUE_VECTOR	1D	w, h, d, t	A pointer to a vector of value data. The location of the vector is specified by the <code>KPDS_VALUE_POSITION</code> attribute. The orientation of a vector can not be changed. You will always get a vector on the <i>e</i> direction. After every get/put pair, or consecutive get or put operations, the position attribute <code>KPDS_VALUE_POSITION</code> will be automatically incremented to the next vector.

Table 1 - Value Primitives

Primitive	Dimension	Default Indexing	Description
KPDS_VALUE_LINE	1D	h, d, t, e	<p>A pointer to a line of value data. The orientation of a line with respect to the Polymorphic Data Model can be changed by using the KPDS_AXIS_ASSIGNMENT attribute. The position index associated with the width axis will be set to zero and the entire length along that axis will be operated on when a get/put is performed. The other attribute that can affect the behavior of a get or put operation involving a line is the KPDS_VALUE_OFFSET attribute. This attribute forces an adjustment of the position in which a unit of data begins. For example, an offset of (1, 0, 0, 0) will cause the beginning of a line to begin at position 1 rather than position zero, and the end of the line to exceed the size along the width axis by one. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_VALUE_POSITION will be automatically incremented to the next line.</p>
KPDS_VALUE_PLANE	2D	d, t, e	<p>A pointer to a plane of value data. The orientation of a value plane with respect to the Polymorphic Data Model is along width and height. What is defined as width and height can be changed using the KPDS_AXIS_ASSIGNMENT attribute. The position indices associated with the width and height axes will be set to zero and the entire length along each axis will be operated on when a get/put is performed. The other attribute that can affect the behavior of a get or put operation involving a plane is the KPDS_VALUE_OFFSET attribute. This attribute forces an adjustment of the position in which a unit of data begins. For example, an offset of (1, 1, 0, 0) will cause the beginning of a plane to begin at position 1 on both the width and height axes rather than position zero, and the end of the plane to exceed the size in both directions direction by one. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_VALUE_POSITION will be automatically incremented to the next plane.</p>

Table 1 - Value Primitives			
Primitive	Dimension	Default Indexing	Description
KPDS_VALUE_VOLUME	3D	t, e	A pointer to a volume of value data. The orientation of a <i>value volume</i> with respect to the Polymorphic Data Model can be changed by using the KPDS_AXIS_ASSIGNMENT attribute. The position indices associated with the width, height, and depth axes will be set to zero and the entire length along each axis will be operated on when a get/put is performed. The other attribute that can affect the behavior of a get or put operation involving a volume is the KPDS_VALUE_OFFSET attribute. This attribute forces an adjustment of the position in which a unit of data begins. For example, an offset of (1, 1, 1, 0, 0) will cause the beginning of a volume to begin at position 1 on the width, height, and depth axes rather than position zero, and the end of the volume to exceed the size in both directions direction by one. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_VALUE_POSITION will be automatically incremented to the next volume.
KPDS_VALUE_REGION	5D	N.A.	A pointer to a region of value data. The upper left-hand corner is specified by the KPDS_VALUE_POSITION and KPDS_VALUE_OFFSET attribute. The size is specified by the KPDS_VALUE_REGION_SIZE attribute. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_VALUE_POSITION will be automatically incremented to the next region.
KPDS_VALUE_ALL	5D	N.A.	A pointer to all of the <i>value</i> data.
KPDS_VALUE_HISTOGRAM	1D	N.A.	A pointer to the histogram for a region of the <i>value</i> data. The upper left-hand corner is specified by the KPDS_VALUE_HIST_POSITION attribute. The size is specified by the KPDS_VALUE_HIST_REGION_SIZE attribute. The number of bins is specified by the KPDS_VALUE_HIST_NUMBINS attribute. The range of the histogram is specified by the KPDS_VALUE_HIST_RANGE attribute.

E.2. Mask Primitives

Table 2 lists the primitives defined by the Polymorphic Data Model for *mask data*. These primitives are stored in and retrieved from the mask data using the `kpds_put_data()` and `kpds_get_data()` functions respectively.

Table 2 - Mask Primitives

Primitive	Dimension	Default Indexing	Description
KPDS_MASK_POINT	0D	w, h, d, t, e	A pointer to a single mask point . The location of the data point is specified by the KPDS_MASK_POSITION attribute. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_MASK_POSITION will be automatically incremented to the next point.
KPDS_MASK_VECTOR	1D	w, h, d, t	A pointer to a vector of mask data. The location of the vector is specified by the KPDS_MASK_POSITION attribute. The orientation of a vector can not be changed. You will always get a vector on the <i>e</i> direction. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_MASK_POSITION will be automatically incremented to the next vector.
KPDS_MASK_LINE	1D	h, d, t, e	A pointer to a line of mask data. The orientation of a line with respect to the Polymorphic Data Model can be changed by using the KPDS_AXIS_ASSIGNMENT attribute. The position index associated with the width axis will be set to zero and the entire length along that axis will be operated on when a get/put is performed. The other attribute that can affect the behavior of a get or put operation involving a line is the KPDS_MASK_OFFSET attribute. This attribute forces an adjustment of the position in which a unit of data begins. For example, an offset of (1, 0, 0, 0, 0) will cause the beginning of a line to begin at position 1 rather than position zero, and the end of the line to exceed the size along the width axis by one. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_MASK_POSITION will be automatically incremented to the next line.

Table 2 - Mask Primitives

Primitive	Dimension	Default Indexing	Description
KPDS_MASK_PLANE	2D	d, t, e	<p>A pointer to a plane of mask data. The orientation of a mask plane with respect to the Polymorphic Data Model is along width and height. What is defined as width and height can be changed using the KPDS_AXIS_ASSIGNMENT attribute. The position indices associated with the width and height axes will be set to zero and the entire length along each axis will be operated on when a get/put is performed. The other attribute that can affect the behavior of a get or put operation involving a plane is the KPDS_MASK_OFFSET attribute. This attribute forces an adjustment of the position in which a unit of data begins. For example, an offset of (1, 1, 0, 0, 0) will cause the beginning of a plane to begin at position 1 on both the width and height axes rather than position zero, and the end of the plane to exceed the size in both directions direction by one. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_MASK_POSITION will be automatically incremented to the next plane.</p>
KPDS_MASK_VOLUME	3D	t, e	<p>A pointer to a volume of mask data. The orientation of a mask volume with respect to the Polymorphic Data Model can be changed by using the KPDS_AXIS_ASSIGNMENT attribute. The position indices associated with the width, height, and depth axes will be set to zero and the entire length along each axis will be operated on when a get/put is performed. The other attribute that can affect the behavior of a get or put operation involving a volume is the KPDS_MASK_OFFSET attribute. This attribute forces an adjustment of the position in which a unit of data begins. For example, an offset of (1, 1, 1, 0, 0) will cause the beginning of a volume to begin at position 1 on the width, height, and depth axes rather than position zero, and the end of the volume to exceed the size in both directions direction by one. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_MASK_POSITION will be automatically incremented to the next volume.</p>

Table 2 - Mask Primitives			
Primitive	Dimension	Default Indexing	Description
KPDS_MASK_REGION	5D	N . A .	A pointer to a region of mask data. The upper left-hand corner is specified by the KPDS_MASK_POSITION and KPDS_MASK_OFFSET attribute. The size is specified by the KPDS_MASK_REGION_SIZE attribute. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_MASK_POSITION will be automatically incremented to the next region.
KPDS_MASK_ALL	5D	N . A .	A pointer to all of the mask data.

E.3. Map Primitives

Table 3 lists the primitives defined by the polymorphic model for *map data*. These primitives are stored in and retrieved from the map data using the `kpds_put_data()` and `kpds_get_data()` functions respectively.

Table 3 - Map Primitives			
Primitive	Dimension	Default Indexing	Description
KPDS_MAP_POINT	0D	mw, mh, me, md, mt	A pointer to a single <i>map point</i> . The location of the data point is specified by the KPDS_MAP_POSITION attribute. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_MAP_POSITION will be automatically incremented to the next point.
KPDS_MAP_VECTOR	1D	mh, me, md, mt	A pointer to a vector of map data. The location of the vector is specified by the KPDS_MAP_POSITION attribute. The orientation of a vector can not be changed. You will always get a vector on the mw direction. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_MAP_POSITION will be automatically incremented to the next vector.

Table 3 - Map Primitives

Primitive	Dimension	Default Indexing	Description
KPDS_MAP_LINE	1D	mh, me, md, mt	<p>A pointer to a line of map data. The orientation of a line with respect to the Polymorphic Data Model can be changed by using the KPDS_AXIS_ASSIGNMENT attribute. The position index associated with the width axis will be set to zero and the entire length along that axis will be operated on when a get/put is performed. The other attribute that can affect the behavior of a get or put operation involving a line is the KPDS_MAP_OFFSET attribute. This attribute forces an adjustment of the position in which a unit of data begins. For example, an offset of (1, 0, 0, 0) will cause the beginning of a line to begin at position 1 rather than position zero, and the end of the line to exceed the size along the width axis by one. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_MAP_POSITION will be automatically incremented to the next line.</p>
KPDS_MAP_PLANE	2D	me, md, mt	<p>A pointer to a plane of map data. The orientation of a <i>map plane</i> with respect to the Polymorphic Data Model is along width and height. What is defined as width and height can be changed using the KPDS_AXIS_ASSIGNMENT attribute. The position indices associated with the width and height axes will be set to zero and the entire length along each axis will be operated on when a get/put is performed. The other attribute that can affect the behavior of a get or put operation involving a plane is the KPDS_MAP_OFFSET attribute. This attribute forces an adjustment of the position in which a unit of data begins. For example, an offset of (1, 1, 0, 0, 0) will cause the beginning of a plane to begin at position 1 on both the width and height axes rather than position zero, and the end of the plane to exceed the size in both directions direction by one. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_MAP_POSITION will be automatically incremented to the next plane.</p>

Table 3 - Map Primitives			
Primitive	Dimension	Default Indexing	Description
KPDS_MAP_VOLUME	3D	md, mt	A pointer to a volume of map data. The orientation of a map volume with respect to the Polymorphic Data Model can be changed by using the KPDS_AXIS_ASSIGNMENT attribute. The position indices associated with the width, height and depth axes will be set to zero and the entire length along each axis will be operated on when a get/put is performed. The other attribute that can affect the behavior of a get or put operation involving a volume is the KPDS_MAP_OFFSET attribute. This attribute forces an adjustment of the position in which a unit of data begins. For example, an offset of (1, 1, 1, 0, 0) will cause the beginning of a volume to begin at position 1 on the width, height, and depth axes rather than position zero, and the end of the volume to exceed the size in both directions direction by one. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_MAP_POSITION will be automatically incremented to the next volume.
KPDS_MAP_REGION	5D	N.A.	A pointer to a region of data. The upper left-hand corner is specified by the KPDS_MAP_POSITION and KPDS_MAP_OFFSET attributes. The size is specified by the KPDS_MAP_REGION_SIZE attribute. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_MAP_POSITION will be automatically incremented to the next region.
KPDS_MAP_ALL	5D	N.A.	A pointer to all of the map data.

E.4. Location Primitives

The different segments of Polymorphic Data Services are organized into arrays of data of up to five dimensions. Elements in these arrays are always accessible by their implicit position within the array. For many applications, such as image processing, this implicit positioning is sufficient. It is enough to know that the elements are arranged into a regular grid, and the size of the grid describes the size of the dataset. However, for some applications, such as satellite image registration, it is important to have a more explicit description of the location of the data points. For example, an image may correspond to an explicit location on the Earth or the pixels may be many meters or even many kilometers across.

The *location segment* is provided to address the need for storing such explicit location information. The location segment can be used to store explicit location vectors which position the value segment in some space. The size of the location segment will match the size of the value segment in the width, height and depth dimension. If multiple volumes are present down the time dimension, each volume will be positioned at the same explicit location through time.

Conceptually, the location segment will contain an explicit location vector for every element vector in the value segment. In practice, storing a single vector for every position along width, height and depth may be more than necessary, depending on the implicit arrangement of the data. Consider the case where all points in the value segment are uniformly spaced across a regular grid. In this case, it would be sufficient to store only a begin and end location to indicate the span of the value segment in explicit space. The explicit location of every vector in the value segment can then be derived based on its implicit position in the value segment. This is the motivation for support of the different location grids in Data Services.

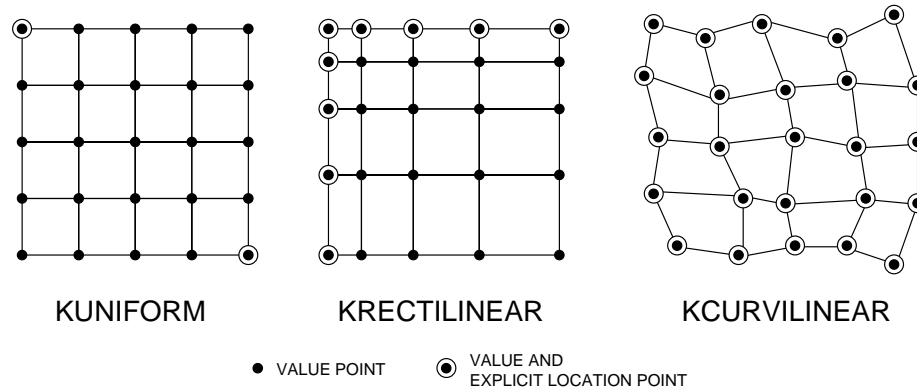


Figure 12: Illustration of *uniform*, *rectilinear* and *curvilinear* location data for the two-dimensional case. The dots represent value segment points, and the circles represent where explicit location information is provided. With each successive type of grid, increasing amounts of explicit location data must be provided. Thus, the dependence on the implicit organization of the data is reduced. Uniform location data consists only of two points, but is entirely dependent on the uniform spacing across the implicit dimensions to have meaning. Rectilinear location data is similar to a uniform grid, except that the spacing along each dimension may vary. Curvilinear location data consists of a specified point for every value segment position, and has no dependence on the implicit organization of the data.

Support is provided for for three different types of location grids. These grid types are UNIFORM, RECTILINEAR, and CURVILINEAR.

- A **uniform** grid consists of two explicit location points which signify a begin and end point which defines a span in explicit space. The explicit position of each point is then derived from the implicit position within the value segment.
- A **rectilinear** grid is similar to a uniform grid, except that the spacing along each dimension may vary. To specify a rectilinear grid, it is necessary to specify the explicit spacing along the width, height, and depth axis. Note that the time segment is actually just a rectilinear specification down the time dimension. The explicit position of each point is formed from the explicit coordinates provided along each implicit axis.
- A **curvilinear** grid has no dependence on the implicit organization of the data. A curvilinear grid requires that a unique explicit position be provided for every implicit position. This has been the standard interface for specifying location data for data services in previous versions. Because of its generality, a curvilinear grid is created by default if no grid is explicitly specified.

Note that since uniform and rectilinear grids are tied to the implicit width-height-depth organization of the data, they are limited to three-space when deriving explicit location information. For convenience, this three-space is said to exist over (x,y,z), with x corresponding to the width dimension, y to the height dimension and z to the depth dimension. The dimension can be defined to be smaller than three, if appropriate for a specific application. For example, a satellite image may just have explicit uniform location information defined over only x and y. If explicit positioning for more than three dimensions, a curvilinear grid must be used.

It is not possible to mix the different types of grids over different implicit dimensions. For example, it is not possible to specify a uniform width and a rectilinear height. When a grid is specified, it will apply to the entire location segment.

E.4.1. Creating Location

The type of location grid must be specified prior to the creation of the location segment using the KPDS_LOCATION_GRID attribute. This attribute can have the value of KUNIFORM, AccuSoftRECTILINEAR, or KCURVILINEAR. This attribute can also have the value of KNONE, when no location segment is present and no grid type has been specified.

The following example illustrates the creation of a uniform location segment :

```
kpds_set_attribute(object, KPDS_LOCATION_GRID, KUNIFORM);
kpds_create_location(object);
```

If no grid type is set and the location segment is created, it will be created as curvilinear by default. Once created, the grid type cannot be changed. The KPDS_LOCATION_GRID attribute should not be set at any time after the location segment has been created. If a different grid type is desired, the location segment should be first destroyed with a call to kpds_destroy_location and then recreated with the new grid type.

E.4.2. Location Primitives

Table 4 lists the primitives defined by the Polymorphic Data Model for *location data*. These primitives are stored in and retrieved from the location data using the kpds_put_data() and kpds_get_data() functions, respectively. There are different primitives provided for accessing the data from each type of location grid.

Table 4a - Curvilinear Location Primitives			
Primitive	Dim'n	Default Indexing	Description
KPDS_LOCATION_POINT	0D	w, h, d, n	A pointer to a single <i>location data point</i> . The location of the data point is specified by the KPDS_LOCATION_POSITION attribute. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_LOCATION_POSITION will be automatically incremented to the next point.

Table 4a - Curvilinear Location Primitives

Primitive	Dim'n	Default Indexing	Description
KPDS_LOCATION_VECTOR	1D	w, h, d	<p>A pointer to a vector of <i>location</i> data. The location of the vector is specified by the KPDS_LOCATION_POSITION attribute. The orientation of a vector cannot be changed. You will always get a vector on the <i>n</i> direction. After every get/put pair or consecutive get or put operations, the position attribute KPDS_LOCATION_POSITION will be automatically incremented to the next vector.</p>
KPDS_LOCATION_LINE	1D	h, d, n	<p>A pointer to a line of location data. The orientation of a line with respect to the Polymorphic Data Model is along width. What is defined as width can be changed using the KPDS_AXIS_ASSIGNMENT attribute. The position index associated with the width axis will be set to zero and the entire length along that axis will be operated on when a get/put is performed.</p> <p>The other attribute that can affect the behavior of a get or put operation involving a line is the KPDS_LOCATION_OFFSET attribute. This attribute forces an adjustment of the position in which a unit of data begins. For example, an offset of (1, 0, 0, 0) will cause the beginning of a line to begin at position 1 rather than position zero, and the end of the line to exceed the size in the along the width axis by one. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_LOCATION_POSITION will be automatically incremented to the next line.</p>

Table 4a - Curvilinear Location Primitives

Primitive	Dim'n	Default Indexing	Description
KPDS_LOCATION_PLANE	2D	d, n	<p>A pointer to a plane of location data. The orientation of a <i>location plane</i> with respect to the Polymorphic Data Model is along width and height. What is defined as width and height can be changed using the KPDS_AXIS_ASSIGNMENT attribute. The position indices associated with the width and height axes will be set to zero and the entire length along each axis will be operated on when a get/put is performed. The other attribute that can affect the behavior of a get or put operation involving a plane is the KPDS_LOCATION_OFFSET attribute. This attribute forces an adjustment of the position in which a unit of data begins. For example, an offset of (1, 1, 0, 0) will cause the beginning of a plane to begin at position 1 on both the width and height axes rather than position zero, and the end of the plane to exceed the size in both directions direction by one. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_LOCATION_POSITION will be automatically incremented to the next plane.</p>
KPDS_LOCATION_VOLUME	3D	n	<p>A pointer to a volume of location data. The orientation of a <i>location volume</i> with respect to the Polymorphic Data Model can be changed by using the KPDS_AXIS_ASSIGNMENT attribute. The position indices associated with the width, height, and depth axes will be set to zero and the entire length along each axis will be operated on when a get/put is performed. The other attribute that can affect the behavior of a get or put operation involving a volume is the KPDS_LOCATION_OFFSET attribute. This attribute forces an adjustment of the position in which a unit of data begins. For example, an offset of (1, 1, 1, 0) will cause the beginning of a volume to begin at position 1 on the width, height, and depth axes rather than position zero, and the end of the volume to exceed the size in both directions direction by one. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_LOCATION_POSITION will be automatically incremented to the next volume.</p>

Table 4a - Curvilinear Location Primitives			
Primitive	Dim'n	Default Indexing	Description
KPDS_LOCATION_REGION	4D	N.A.	A pointer to a region of location data. The upper left-hand corner is specified by the KPDS_LOCATION_POSITION and KPDS_LOCATION_OFFSET attributes. The size is specified by the KPDS_LOCATION_REGION_SIZE attribute. After every get/put pair, or consecutive get or put operations, the position attribute KPDS_LOCATION_POSITION will be automatically incremented to the next region.
KPDS_LOCATION_ALL	4D	N.A.	A pointer to all of the location data.

Rectilinear location data has primitives which allow the specification of an explicit array of data down each dimension. These primitives are specified over the location width, height and depth dimensions. In essence, these primitives are analogous to the primitives for the time segment. The regular location position, offset and size attributes are used with rectilinear location data, with the width component affecting the width primitives, the height component affecting the height primitives and the depth component affecting the depth primitives. Data for each dimension must be stored or retrieved with a separate put_data or get_data call and the corresponding primitive for that dimension.

Table 4b - Rectilinear Location Primitives			
Primitive	Dim'n	Default Indexing	Description
KPDS_LOCATION_WIDTH_POINT KPDS_LOCATION_HEIGHT_POINT KPDS_LOCATION_DEPTH_POINT	0D	w, h, d, n	A pointer to a single rectilinear location data point. The location of the data point is specified by the corresponding component of the KPDS_LOCATION_POSITION attribute. The dimension component of the position attribute is not relevant for rectilinear location data.
KPDS_LOCATION_WIDTH_REGION KPDS_LOCATION_HEIGHT_REGION KPDS_LOCATION_DEPTH_REGION	1D	N.A.	A pointer to a region of rectilinear location data. The upper left-hand corner is specified by the KPDS_LOCATION_POSITION and KPDS_LOCATION_OFFSET attributes. The size is specified by the KPDS_LOCATION_REGION_SIZE attribute.
KPDS_LOCATION_WIDTH_ALL KPDS_LOCATION_HEIGHT_ALL KPDS_LOCATION_DEPTH_ALL	1D	N.A.	A pointer to all of the rectilinear location data along a certain dimension.

Uniform location data actually does not have any primitives associated with it. The explicit begin and end points are specified via the attributes KPDS_LOCATION_BEGIN and KPDS_LOCATION_END. These attributes are only available if uniform location data is present. A uniform location segment must be created before these attributes can be set.

Uniform Location Attributes		
Attribute and Default	Legal Values	Definition
KPDS_LOCATION_BEGIN double w 0.0 h 0.0 d 0.0		This attribute represents an explicit begin marker point for the Polymorphic Data Model. This begin point maps to the implicit origin of the data model. This attribute can only be set if uniform <i>location</i> data has been explicitly created with a <code>kpds_create_location</code> call. See also: <code>KPDS_LOCATION_END</code> . Persistence: permanent
KPDS_LOCATION_END double w 0.0 h 0.0 d 0.0		This attribute represents an explicit end marker point for the Polymorphic Data Model. This end point maps to the implicit extent of the data model. This attribute can only be set if uniform <i>location</i> data has been explicitly created with a <code>kpds_create_location</code> call. See also: <code>KPDS_LOCATION_END</code> . Persistence: permanent

E.4.3. Presentation of Location Data

In order to minimize the complexity of processing incoming location data, functionality has been provided which can present any type of location grid as curvilinear location data. Unlike any other presentation capabilities in Data Services, it is not necessary to set any presentation attributes to invoke this capability. It is embedded into the primitive access routines and will automatically be invoked on a `kpds_get_data` call.

To process location data, a program minimally can just get curvilinear location primitives. If the location segment actually consists of uniform or rectilinear location data, it will be presented back through the curvilinear primitives as if the data actually was stored as curvilinear. If only uniform or rectilinear location data is present, the contents of that vector will automatically be constructed and returned via the `kpds_get_data` call. So, for example, it is always possible to get any given curvilinear location vector using the `KPDS_LOCATION_VECTOR` primitive, regardless of the grid type of the data being stored.

In general, any more explicit form of location data can be retrieved when a less explicit form is present. Thus, this presentation capability works for rectilinear location data as well. Uniform data can be retrieved, if desired, as rectilinear location data. Note that it is never possible to retrieve a less explicit form of location data when a more explicit form is present. For example, you could never retrieve uniform location primitives from curvilinear location data. Furthermore, this presentation functionality also only works for data retrieval, not for data storage. Clearly, it is not possible to store a set of explicit curvilinear location points as uniform data because it may not actually be uniformly spaced.

E.5. Time Primitives

Table 5 lists the primitives defined by the polymorphic model for *time data*. These primitives are stored in and retrieved from the time data using the `kpds_put_data()` and `kpds_get_data()` functions respectively.

Table 5 - Time Primitives			
Primitive	Dimension	Default Indexing	Description
KPDS_TIME_POINT	0D	t	A pointer to a single <i>time point</i> . The location of the data point is specified by the <code>KPDS_TIME_POSITION</code> attribute. After every get/put pair, or consecutive get or put operations, the position attribute <code>KPDS_TIME_POSITION</code> will be automatically incremented to the next point.
KPDS_TIME_REGION	1D	N.A.	A pointer to a region of time data. The starting point is specified by the <code>KPDS_TIME_POSITION</code> and <code>KPDS_TIME_OFFSET</code> attributes. The size is specified by the <code>KPDS_TIME_REGION_SIZE</code> attribute. After every get/put pair, or consecutive get or put operations, the position attribute <code>KPDS_TIME_POSITION</code> will be automatically incremented to the next region.
KPDS_TIME_ALL	1D	N.A.	A pointer to all of the time data.

F. Attributes Defined by the Polymorphic Data Model

Table 6 lists the attributes defined by the Polymorphic Data Model and Table 7 describes these attributes in detail. Each attribute is associated with a particular component of the data object, such as the mask. The attributes are stored and retrieved using `kpds_set_attribute()` and `kpds_get_attribute()` respectively.

```
kpds_set_attribute(obj2, KPDS_VALUE_SIZE, width, height, depth, time, elements);
kpds_get_attribute(obj1, KPDS_VALUE_SIZE, &width, &height, &depth, &time, &elements);
```

The `kpds_get_attribute()` and `kpds_set_attribute()` functions have variable argument lists as specified in the data model attribute table that follows in Table 6.

The last three functions listed — `kpds_get_attributes()`, `kpds_set_attributes()`, and `kpds_match_attributes()` — are multiple attribute functions, and the argument lists for these *must* be NULL terminated. See the corresponding function descriptions in Section G for more information on usage.

Each column in the tables that follow are defined as follows:

Attribute and Default — This is the attribute name and the data type of the attribute's value(s) and suggested variable names to use for multi-variable attributes, like `KPDS_VALUE_SIZE`. The attribute order is given, and the default value for the variable(s). If the default is *read only* then you cannot set or change the corresponding attribute, but only read it. If the default is *unknown*, then you can set an input object to the stored value(s), or you must set the attribute for objects created via `kpds_create_object` or output objects.

Legal Values — Where appropriate, a list of preprocessor symbols or a numerical range is given that indicates the legal range of values for the attribute.

Description — A description of the attribute and how it should be used.

Persistence — This field indicates whether the attributes are *stored* when written to a file, or *transient* (not stored) with the data. If the attribute is stored, then when the transport is re-opened, the value of the attribute will be restored. If the attribute is transient, then it is only valid during the current processing of the data, and when the object is opened, it is set to the default value.

F.1. Global Attributes

Global Attributes		
Attribute and Default	Legal Values	Definition
KPDS_ARCHITECTURE int architecture	KMACH_UNKNOWN KMACH_LITTLE_ENDIAN_IEEE KMACH_LITTLE_ENDIAN_VAX KMACH_LITTLE_ENDIAN_64 KMACH_BIG_ENDIAN_IEEE KMACH_BIG_ENDIAN_CRAY	This attribute is an integer value which encodes a description of the floating point and integer representation for the machine which what used to generate the object. A set of C defines are typically used when operating on the value of this attribute in a program. Typically, this attribute is set based on an examination of the input object, and is set to the local architecture on an output object. The encoding scheme and specific values for these defines can be found in \$BUILD/include/machine/kmachine.h. Persistence: stored
KPDS_AXIS_ASSIGNMENT int w KWIDTH h KHEIGHT d KDEPTH	KWIDTH KHEIGHT KDEPTH	This attribute allows the width, height, and depth axes to be reassigned to one-another to simplify visualization or processing. The effect of reassigning axes is similar to transposing a matrix. Persistence: stored
KPDS_COMMENT char * comment NULL		This attribute is a NULL terminated string used to document the object. This attribute is used by a user or programmer to describe the origin or nature of the data set. When this attribute is set, it overwrites anything previously held in this attribute. Therefore, it is up to the programmer to first get the comment attribute, append new information to it, and then set the entire comment attribute, if prior comment information is to be propagated. To clear the comment attribute, pass in NULL when setting the attribute. This attribute is copied with the <code>kpds_copy_object()</code> and <code>kpds_copy_object_attr()</code> calls. Persistence: stored

Global Attributes		
Attribute and Default	Legal Values	Definition
KPDS_COUPLING int coupling See Note 1.	KCOUPLED KDEMAND KUNCOUPLED	<p>When this attribute is set to KCOUPLED, changes to any attribute that affects the physical representation of the data (for example, data type, size, etc.) will be propagated to the physical layer immediately. Otherwise, the presentation layer is the only layer that is changed, --the physical layer remains unchanged. The difference between KUNCOUPLED and KDEMAND is that KDEMAND allows the <code>kpds_sync_object()</code> function call to force an update of the presentation and physical layers. When this attribute is set to KUNCOUPLED, the calling the <code>kpds_sync_object()</code> will not do anything. See <code>kpds_sync_object()</code> for more information.</p> <p>Persistence: transient</p>
KPDS_DATA_FORMAT int data_format	KUBYTE KUSHORT KFLOAT KUSERDEFINED	<p>This attribute defines the format of data to be expected in terms of predefined theoretical ranges. The actual range of data may be within this range or outside of the range.</p> <p>Persistence: stored</p>
KPDS_DATA_RANGE double min max		<p>This attribute defines the theoretical range of data to be expected. The actual range of data may be within this range or outside of the range.</p> <p>Persistence: stored</p>
KPDS_DATE char * date current date		<p>This attribute is a NULL terminated string used to record the date of the creation of the data object. This attribute is NOT copied by <code>kpds_copy_attributes()</code>. To assign the current date as defined by computer system, pass in NULL when setting the attribute. The date will be stored in the default format of the UNIX <code>date</code> command ("day month date HH:MM:SS timezone year", e.g. "Wed Mar 10 00:07:23 MST 1994")</p> <p>Persistence: stored</p>

Global Attributes		
Attribute and Default	Legal Values	Definition
KPDS_FORMAT char * format viff	kdf viff jpeg pnm pcx xpm xbm xwd eps rast avs ascii raw	<p>This attribute specifies the file format that will be used with the object. If the object is an input object, then this attribute is automatically initialized to the file format that the object is stored in. If the object is an output object, then this attribute defaults to "viff", indicating that the output data file will be a viff. On output objects, this attribute can be set to any of the legal values. The result is that when the object is closed, it will be written out in the format specified.</p> <p>Persistence: stored</p>
KPDS_FORMAT_DESCRIPTION char * description viff	N/A	<p>This (read-only) attribute retrieves the file format description that will describes the format associated with an object.</p> <p>Persistence: transient</p>
KPDS_HISTORY char * history NULL		<p>This attribute is a string that describes the operations that have been performed on the original data set that result in the current data set. Typically, programs names and their command line arguments are listed here to reproduce this data set. This attribute will only be set if the KHOROS_HISTORY environment variable is set.</p> <p>Persistence: stored</p>
KPDS_HISTORY_MODE int hmode	KAPPEND_HISTORY KREPLACE_HISTORY	<p>This attribute determines how the KPDS_HISTORY attribute is interpreted when being set. If the history mode is set to KAPPEND_HISTORY, then the string begin set will be appended to the existing history string. If the history mode is set to KREPLACE_HISTORY, then the string being set will replace the existing history string.</p> <p>Persistence: transient</p>
KPDS_KERNEL_ORIGIN int w 0 h 0 d 0 t 0 e 0		<p>This attribute is used to specify a "hot spot" in the data set that is interpreted as the center point of a 5 dimensional convolution kernel.</p>

Global Attributes		
Attribute and Default	Legal Values	Definition
		Persistence: stored
KPDS_MAPPING_MODE int mapping_mode KUNMAPPED	KMAPPED KUNMAPPED	This attribute specifies how to interpret the <i>map</i> data if it is present. The default interpretation KUNMAPPED means that nothing is done with the map. It is then the responsibility of the programmer to interpret or ignore the map. The KMAPPED mode causes the <i>map</i> and <i>value</i> data to be merged. This means that the <i>map</i> will not appear to be present. Instead, the <i>value</i> data will assimilate the attributes of the <i>map</i> data as appropriate. Furthermore, any <i>value</i> data that is retrieved via a call to <code>kpds_get_data</code> will be mapped through the <i>map</i> before being returned. For example, the KPDS_VALUE_SIZE attribute will have its KELEMENTS value multiplied by the KMAP_WIDTH size of the <i>map</i> . The KPDS_VALUE_DATA_TYPE will actually take the value of the KPDS_MAP_DATA_TYPE. Each value will be used as an index into the <i>map</i> and used as the KMAP_HEIGHT index. All get and and set attribute calls will behave in this manner.
		Persistence: transient
KPDS_MASKED_VALUE_PRESENTATION int mask_mode KUSE_ORIGINAL	KUSE_ORIGINAL KUSE_SUBSTITUTE_VALUE	This attribute takes on one of two values: either KUSE_ORIGINAL, or KUSE_SUBSTITUTE_VALUE. If set to KUSE_ORIGINAL, then regardless of the <i>mask</i> value at a given data point, the value returned for that data point is what is stored there. If the attribute is set to KUSE_SUBSTITUTE_VALUE, then the KPDS_MASK_SUBSTITUTE_VALUE will replace any <i>value</i> data point that has a 0 mask.
		Persistence: transient
KPDS_MASK_SUBSTITUTE_VALUE double real 1.0 imag 0.0		This value is used to replace <i>value</i> data whose mask indicates that it is invalid data. It is only used when the KPDS_MASKED_VALUE_PRESENTATION is set to KUSE_SUBSTITUTE_VALUE.
		Persistence: stored
KPDS_NAME char * name		This attribute is used to obtain the filename associated with the specified data object. This is that name passed in to <code>kpds_open_object</code> , <code>kpds_open_output_object</code> , or <code>kpds_open_input_object</code> . Objects that are instantiated with <code>kpds_create_object</code> do not have a filename. In such instances, this attribute's value is NULL.
		Persistence: stored

Global Attributes		
Attribute and Default	Legal Values	Definition
KPDS_POINT_SIZE double w 1.0 h 1.0 d 1.0 t 1.0 e 1.0	> 0.0	This attribute indicates the physical dimension in world coordinates of a point in the data set. A single sampled point represents a continuous volume of data in world coordinate space. This attribute indicates the size of that volume. Persistence: stored
KPDS_SUBOBJECT_POSITION int w 0 h 0 d 0 t 0 e 0		This is the offset of the current data object in a parent object. Typically the value of this attribute will be {0, 0, 0, 0, 0}, but if this object was extracted from a "parent object", via kextract or other means, then this attribute will indicate the position in the parent object from which this region was extracted. It is intended to be used to automate the process of reinserting the object into its parent once region-of-interest processing is complete. Persistence: stored

F.2. Value Segment Attributes

Value Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_VALUE_COMPLEX_CONVERT int convert KREAL	KIMAGINARY KMAGNITUDE KPHASE KREAL KMAGSQ KMAGSQP1 KLOGMAG KLOGMAGP1	<p>This attribute specifies how complex data should be converted. If it is converted to a "lower" data type, this attribute specifies how to down-convert the data. For example if the data is actually complex, but the presentation attribute is byte, the complex data would first be converted to the representation defined by this attribute, and then converted to byte.</p> <p>If the data is being converted from a "lower" data type to a complex data type, this attribute defines how the data should be interpreted — as the real or imaginary component of the complex pair. KPHASE and KMAGNITUDE are invalid values for up converting to complex, and will result in an error.</p> <p>Persistence: transient</p>
KPDS_VALUE_DATA_TYPE int data_type	KBIT KBYTE KUBYTE KSHORT KUSHORT KINT KUINT KLONG KULONG KFLOAT KDOUBLE KCOMPLEX KDCOMPLEX	<p>This attribute is used to get or set the data type, or numerical representation of the data. This data type will be the presentation data type, not necessarily the physical data type. See the KPDS_COUPLING attribute for more information on how to control the presentation and physical data types. When the application programmer specifies a presentation data type that is different than the actual data type of the stored data, the get kpds_get_data function will convert the data to return the requested data type. Likewise, the kpds_put_data function expects data that is in the data type specified by this attribute to the output object, and if the data being "put" is of a different type, it will be converted. This attribute must be set for objects created via kpds_create_object or output objects that are opened with kpds_open_output_object or kpds_open_object, or else the get and put data calls will fail.</p> <p>Persistence: stored</p>

Value Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_VALUE_INCREMENT_SIZE int width height depth num_volumes		<p>This attribute is used to alter how the auto-increment state machine behaves. Normally, the size of the data set is used to dictate how position auto-advances from one position to the next, based on the primitive being accessed. This attribute, if set, will be used instead of the size of the data set for controlling auto-advancement. One place where such functionality is useful is when a smaller data set is being inserted into a larger one. The larger destination data set's INCREMENT_SIZE attribute can be set to the size of the smaller source data set so that the auto-advancement stays synchronized across all dimensions.</p> <p>Persistence: transient</p>
KPDS_VALUE_INTERPOLATE int interpolate KPAD	KNONE KPAD KZERO_ORDER KWRAP	<p>This attribute specifies how the data should be presented if the application program requests a size different from what is physically stored. If the size requested is larger than the physical size and the interpolation requested is KPAD the pad value will be returned for all points outside of the physical size. If the size requested is smaller than the physical size and the interpolation requested is KPAD the returned data is clipped to the size requested. If the size requested is larger than the physical size and the interpolation requested is KZERO_ORDER the data is duplicated. If the size requested is smaller than the physical size and the interpolation requested is KZERO_ORDER the data is sub-sampled. If the interpolation requested is KWRAP then the size change will be resolved by duplicating the data set. If KWRAP is set, then out-of-bounds data accesses will also be filled with duplicated data. If the interpolate attribute is set to KNONE, an error will be returned if the program requests a size different from what is physically stored.</p> <p>Persistence: transient</p>
KPDS_VALUE_LINE_INFO int line_size num_lines		<p>This attribute will return the number of points in a line and the number of lines in the dataset. The line size will be the size of the width axis and the of lines number will be the product of the other axes' sizes.</p> <p>Persistence: transient</p>
KPDS_VALUE_NORM_MAX double norm_max	> norm_min	<p>This attribute specifies the maximum to be used when normalizing data values. This attribute is used in conjunction with the KPDS*_NORM_MIN attribute, respectively, to determine the bounds of a normalization operation. This attribute comes into play when the KPDS*_SCALING attribute is set to KNORMALIZE.</p>

Value Segment Attributes		
Attribute and Default	Legal Values	Definition
		Persistence: transient
KPDS_VALUE_NORM_MIN double norm_min	< norm_max	This attribute specifies the minimum to be used when normalizing data values. This attribute is used in conjunction with the KPDS*_NORM_MAX attribute, respectively, to determine the bounds of a normalization operation. This attribute comes into play when the KPDS*_SCALING attribute is set to KNORMALIZE.
		Persistence: transient
KPDS_VALUE_OFFSET int offset_w 0 offset_h 0 offset_d 0 offset_t 0 offset_e 0	+/- int	These attribute values specify the offset into the data position for all primitives. Offset values beyond the boundaries of the data are valid.
		Persistence: transient
KPDS_VALUE_OPTIMAL_REGION_SIZE int region_width region_height region_depth region_time region_elements number_of_regions	> 0	This attribute will return the size of a region of data and the number of such regions that is most efficient to process in terms of performance and memory use. The KPDS*_REGION_SIZE attribute controls the size and dictates the number of regions, which will always be rounded up. See KPDS*_REGION_SIZE for more information.
		Persistence: transient
KPDS_VALUE_PAD_VALUE double real_value 0.0 imag_value 0.0		This attribute specifies the real (and imaginary) values of the pad data if the KPDS*_INTERPOLATE attribute is set to KPAD, respectively. The double values must be specified, whether the data is real or complex. The pad values will internally be converted from double to the appropriate data type. The default pad value for <i>location</i> , <i>map</i> , <i>time</i> and <i>value</i> is 0. The default pad value for the <i>mask</i> is 1 because when padding data, the padded portion of the <i>value</i> data should initially be considered valid.
		Persistence: transient

Value Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_VALUE_PLANE_INFO int plane_width plane_height num_planes		This attribute will return the size of a plane of data in points and the number of planes in the dataset. The plane size will be the size of the width and height axes and the number will be the product of the sizes of the other axes. Persistence: transient
KPDS_VALUE_POSITION int w 0 h 0 d 0 t 0 e 0	+/- int	The position attribute specifies four indices to locate a specific PRIMITIVE in the <i>location</i> data. w is <i>width</i> , h is <i>height</i> , d is <i>depth</i> and n is <i>dimension</i> . Persistence: transient
KPDS_VALUE_REGION_INFO int region_width region_height region_depth region_time region_elements num_regions		This attribute will return the size of a region of data in points and the number of regions in the data. The KPDS_*_REGION_SIZE attribute controls the size and dictates the number of regions, which will always be rounded up. See KPDS_*_REGION_SIZE for more information. Persistence: transient
KPDS_VALUE_REGION_SIZE int region_width 1 region_height 1 region_depth 1 region_time 1 region_elements 1	> 0	These attribute values specify the size of the region being processed. If the region size for one or more dimensions is not a even multiple of the data size, then the pad value will be returned by <code>kpds_get_data</code> for all data outside of the data space, which is set by <code>KPDS_*_PAD_VALUE</code> . On a <code>kpds_put_data</code> call, data points outside of the data space will be truncated. E.g. if the object width is 512 and the region width is 200, then getting the first two regions will return the data for those regions. The next get will return the remaining 112 points from the data in the width direction with the remaining 88 points set to the pad value. On a put for this same setup, the first two puts will place full regions into the data object, but the last put will place only the first 112 points into the data object in the width direction and the last 88 points are truncated. Persistence: transient

Value Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_VALUE_SCALE_FACTOR double scale_factor 1.0		This attribute specifies the scaling factor to be used when scaling data values. This attribute comes into play when the KPDS_*_SCALING attribute is set to KSCALE, respectively. Persistence: transient
KPDS_VALUE_SCALE_OFFSET double offset_real 0.0 offset_imaginary 0.0		This attribute specifies the scaling offset to be used when scaling data values. This attribute comes into play when the KPDS_*_SCALING attribute is set to KSCALE, respectively. Persistence: transient
KPDS_VALUE_SCALING int scaling KNONE	KNONE KNORMALIZE KSCALE	This attribute specifies whether scaling or normalization should be performed. If KSCALE is specified for the <i>value</i> data, values will be scaled, according to the KPDS_VALUE_SCALE_FACTOR and KPDS_VALUE_SCALE_OFFSET attributes. If KNORMALIZE is specified for the <i>value</i> data, values will be normalized using the KPDS_VALUE_NORM_MIN and KPDS_VALUE_NORM_MAX attributes. If this attribute is set to KNONE for the value data, values will not be scaled or normalized. The same is true for the <i>map</i> and <i>mask</i> data. They will use their respective scale factor & offset and normalize minimum & maximum attributes. Persistence: transient
KPDS_VALUE_SIZE int width height depth time elements	> 0	This attribute specifies the size of the dimensions <i>width</i> , <i>height</i> , <i>depth</i> , <i>elements</i> , <i>time</i> , <i>location dimension</i> , <i>map width</i> , <i>map height</i> , <i>map elements</i> , <i>map depth</i> <i>map time</i> . When the application programmer specifies a size larger than the actual size of stored data, the get functions will sub-sampled, clipped, padded or duplicated the data to present the program with the requested amount, see the attribute KPDS_VALUE_INTERPOLATE for more details. The put functions store exactly the size that the physical attributes will allow even if the amount of data "put" (set by the presentation attributes) is different. This attribute must be set for objects created via <code>kpds_create_object</code> or output objects else get/put data calls will fail. The size of the <i>mask</i> and <i>value</i> data is identical. The <i>time</i> size is shared between the <i>time</i> , <i>mask</i> and <i>value</i> data. The <i>width</i> , <i>height</i> <i>depth</i> are shared between the <i>location</i> , <i>mask</i> <i>value</i> data.

Value Segment Attributes		
Attribute and Default	Legal Values	Definition
		Persistence: transient
KPDS_VALUE_VECTOR_INFO int plane_size num_vectors		This attribute will return the number of points in a vector of data and the number of vectors in the dataset. The vector vector definition for the data primitive. For KPDS_VALUE_VECTOR_INFO, the size is the size of the element vector and the number is the product of the remaining dimensions.
		Persistence: transient
KPDS_VALUE_VOLUME_INFO int width height depth num_volumes		This attribute will return the size of a volume of data in points and the number of volumes in the dataset. The volume size will be the size of the width, height, and depth axes. The number volumes will be the product of the sizes of the remaining axes.
		Persistence: transient

F.3. Mask Segment Attributes

Mask Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_MASK_DATA_TYPE int data_type	KBIT KBYTE KUBYTE KSHORT KUSHORT KINT KUINT KLONG KULONG KFLOAT KDOUBLE KCOMPLEX KDCOMPLEX	<p>This attribute is used to get or set the data type, or numerical representation of the data. This data type will be the presentation data type, not necessarily the physical data type. See the <code>KPDS_COUPLING</code> attribute for more information on how to control the presentation and physical data types. When the application programmer specifies a presentation data type that is different than the actual data type of the stored data, the <code>get kpds_get_data</code> function will convert the data to return the requested data type. Likewise, the <code>kpds_put_data</code> function expects data that is in the data type specified by this attribute to the output object, and if the data being "put" is of a different type, it will be converted. This attribute must be set for objects created via <code>kpds_create_object</code> or output objects that are opened with <code>kpds_open_output_object</code> or <code>kpds_open_object</code>, or else the get and put data calls will fail.</p> <p>Persistence: stored</p>
KPDS_MASK_INCREMENT_SIZE int width height depth num_volumes		<p>This attribute is used to alter how the auto-increment state machine behaves. Normally, the size of the data set is used to dictate how position auto-advances from one position to the next, based on the primitive being accessed. This attribute, if set, will be used instead of the size of the data set for controlling auto-advancement. One place where such functionality is useful is when a smaller data set is being inserted into a larger one. The larger destination data set's <code>INCREMENT_SIZE</code> attribute can be set to the size of the smaller source data set so that the auto-advancement stays synchronized across all dimensions.</p> <p>Persistence: transient</p>

Mask Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_MASK_INTERPOLATE int interpolate KPAD	KNONE KPAD KZERO_ORDER KWRAP	<p>This attribute specifies how the data should be presented if the application program requests a size different from what is physically stored. If the size requested is larger than the physical size and the interpolation requested is KPAD the pad value will be returned for all points outside of the physical size. If the size requested is smaller than the physical size and the interpolation requested is KPAD the returned data is clipped to the size requested. If the size requested is larger than the physical size and the interpolation requested is KZERO_ORDER the data is duplicated. If the size requested is smaller than the physical size and the interpolation requested is KZERO_ORDER the data is sub-sampled. If the interpolation requested is KWRAP then the size change will be resolved by duplicating the data set. If KWRAP is set, then out-of-bounds data accesses will also be filled with duplicated data. If the interpolate attribute is set to KNONE, an error will be returned if the program requests a size different from what is physically stored.</p> <p>Persistence: transient</p>
KPDS_MASK_LINE_INFO int line_size num_lines		<p>This attribute will return the number of points in a line and the number of lines in the dataset. The line size will be the size of the width axis and the of lines number will be the product of the other axes' sizes.</p> <p>Persistence: transient</p>
KPDS_MASK_OFFSET int offset_w 0 offset_h 0 offset_d 0 offset_t 0 offset_e 0	+/- int	<p>These attribute values specify the offset into the data position for all primitives. Offset values beyond the boundaries of the data are valid.</p> <p>Persistence: transient</p>
KPDS_MASK_OPTIMAL_REGION_SIZE int region_width region_height region_depth region_time region_elements number_of_regions	> 0	<p>This attribute will return the size of a region of data and the number of such regions that is most efficient to process in terms of performance and memory use. The KPDS_*_REGION_SIZE attribute controls the size and dictates the number of regions, which will always be rounded up. See KPDS_*_REGION_SIZE for more information.</p>

Mask Segment Attributes		
Attribute and Default	Legal Values	Definition
		Persistence: transient
KPDS_MASK_PAD_VALUE double pad_value 1.0		This attribute specifies the real (and imaginary) values of the pad data if the KPDS_*_INTERPOLATE attribute is set to KPAD, respectively. The double values must be specified, whether the data is real or complex. The pad values will internally be converted from double to the appropriate data type. The default pad value for <i>location</i> , <i>map</i> , <i>time</i> and <i>value</i> is 0. The default pad value for the <i>mask</i> is 1 because when padding data, the padded portion of the <i>value</i> data should initially be considered valid.
		Persistence: transient
KPDS_MASK_PLANE_INFO int plane_width plane_height num_planes		This attribute will return the size of a plane of data in points and the number of planes in the dataset. The plane size will be the size of the width and height axes and the number will be the product of the sizes of the other axes.
		Persistence: transient
KPDS_MASK_POSITION int w 0 h 0 d 0 t 0 e 0	+/- int	The position attribute specifies four indices to locate a specific PRIMITIVE in the <i>location</i> data. w is <i>width</i> , h is <i>height</i> , d is <i>depth</i> and n is <i>dimension</i> .
		Persistence: transient
KPDS_MASK_REGION_INFO int region_width region_height region_depth region_time region_elements num_regions		This attribute will return the size of a region of data in points and the number of regions in the data. The KPDS_*_REGION_SIZE attribute controls the size and dictates the number of regions, which will always be rounded up. See KPDS_*_REGION_SIZE for more information.
		Persistence: transient

Mask Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_MASK_REGION_SIZE int region_width 1 region_height 1 region_depth 1 region_time 1 region_elements 1	> 0	<p>These attribute values specify the size of the region being processed. If the region size for one or more dimensions is not an even multiple of the data size, then the pad value will be returned by <code>kpds_get_data</code> for all data outside of the data space, which is set by <code>KPDS*_PAD_VALUE</code>. On a <code>kpds_put_data</code> call, data points outside of the data space will be truncated. E.g. if the object width is 512 and the region width is 200, then getting the first two regions will return the data for those regions. The next get will return the remaining 112 points from the data in the width direction with the remaining 88 points set to the pad value. On a put for this same setup, the first two puts will place full regions into the data object, but the last put will place only the first 112 points into the data object in the width direction and the last 88 points are truncated.</p> <p>Persistence: transient</p>
KPDS_MASK_SIZE int width height depth time elements	> 0	<p>This attribute specifies the size of the dimensions <i>width</i>, <i>height</i>, <i>depth</i>, <i>elements</i>, <i>time</i>, <i>location dimension</i>, <i>map width</i>, <i>map height</i>, <i>map elements</i>, <i>map depth</i> <i>map time</i>. When the application programmer specifies a size larger than the actual size of stored data, the get functions will sub-sampled, clipped, padded or duplicated the data to present the program with the requested amount, see the attribute <code>KPDS_VALUE_INTERPOLATE</code> for more details. The put functions store exactly the size that the physical attributes will allow even if the amount of data "put" (set by the presentation attributes) is different. This attribute must be set for objects created via <code>kpds_create_object</code> or output objects else get/put data calls will fail.</p> <p>The size of the <i>mask</i> and <i>value</i> data is identical. The <i>time</i> size is shared between the <i>time</i>, <i>mask</i> and <i>value</i> data. The <i>width</i>, <i>height</i> <i>depth</i> are shared between the <i>location</i>, <i>mask</i> <i>value</i> data.</p> <p>Persistence: stored</p>
KPDS_MASK_VECTOR_INFO int vector_length num_vectors		<p>This attribute will return the number of points in a vector of data and the number of vectors in the dataset. The vector vector definition for the data primitive. For <code>KPDS_VALUE_VECTOR_INFO</code>, the size is the size of the element vector and the number is the product of the remaining dimensions.</p> <p>Persistence: transient</p>

Mask Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_MASK_VOLUME_INFO int width height depth num_volumes		This attribute will return the size of a volume of data in points and the number of volumes in the dataset. The volume size will be the size of the width, height, and depth axes. The number volumes will be the product of the sizes of the remaining axes.
		Persistence: transient

F.4. Map Segment Attributes

Map Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_MAP_COMPLEX_CONVERT int convert KREAL	KIMAGINARY KMAGNITUDE KPHASE KREAL KMAGSQ KMAGSQP1 KLOGMAG KLOGMAGP1	<p>This attribute specifies how complex data should be converted. If it is converted to a "lower" data type, this attribute specifies how to down-convert the data. For example if the data is actually complex, but the presentation attribute is byte, the complex data would first be converted to the representation defined by this attribute, and then converted to byte.</p> <p>If the data is being converted from a "lower" data type to a complex data type, this attribute defines how the data should be interpreted — as the real or imaginary component of the complex pair. KPHASE and KMAGNITUDE are invalid values for up converting to complex, and will result in an error.</p> <p>Persistence: transient</p>
KPDS_MAP_DATA_TYPE int datatype	KBIT KBYTE KUBYTE KSHORT KUSHORT KINT KUINT KLONG KULONG KFLOAT KDOUBLE KCOMPLEX KDComplex	<p>This attribute is used to get or set the data type, or numerical representation of the data. This data type will be the presentation data type, not necessarily the physical data type. See the KPDS_COUPLING attribute for more information on how to control the presentation and physical data types. When the application programmer specifies a presentation data type that is different than the actual data type of the stored data, the get kpds_get_data function will convert the data to return the requested data type. Likewise, the kpds_put_data function expects data that is in the data type specified by this attribute to the output object, and if the data being "put" is of a different type, it will be converted. This attribute must be set for objects created via kpds_create_object or output objects that are opened with kpds_open_output_object or kpds_open_object, or else the get and put data calls will fail.</p> <p>Persistence: stored</p>

Map Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_MAP_INCREMENT_SIZE int width height depth time elements num_volumes		This attribute is used to alter how the auto-increment state machine behaves. Normally, the size of the data set is used to dictate how position auto-advances from one position to the next, based on the primitive being accessed. This attribute, if set, will be used instead of the size of the data set for controlling auto-advancement. One place where such functionality is useful is when a smaller data set is being inserted into a larger one. The larger destination data set's INCREMENT_SIZE attribute can be set to the size of the smaller source data set so that the auto-advancement stays synchronized across all dimensions. Persistence: transient
KPDS_MAP_INTERPOLATE int interpolate KPAD	KNONE KPAD KZERO_ORDER KWRAP	This attribute specifies how the data should be presented if the application program requests a size different from what is physically stored. If the size requested is larger than the physical size and the interpolation requested is KPAD the pad value will be returned for all points outside of the physical size. If the size requested is smaller than the physical size and the interpolation requested is KPAD the returned data is clipped to the size requested. If the size requested is larger than the physical size and the interpolation requested is KZERO_ORDER the data is duplicated. If the size requested is smaller than the physical size and the interpolation requested is KZERO_ORDER the data is sub-sampled. If the interpolation requested is KWRAP then the size change will be resolved by duplicating the data set. If KWRAP is set, then out-of-bounds data accesses will also be filled with duplicated data. If the interpolate attribute is set to KNONE, an error will be returned if the program requests a size different from what is physically stored. Persistence: transient
KPDS_MAP_LINE_INFO int line_size num_lines		This attribute will return the number of points in a line and the number of lines in the dataset. The line size will be the size of the width axis and the of lines number will be the product of the other axes' sizes. Persistence: transient
KPDS_MAP_NORM_MAX double norm_max	> norm_min	This attribute specifies the maximum to be used when normalizing data values. This attribute is used in conjunction with the KPDS_*_NORM_MIN attribute, respectively, to determine the bounds of a normalization operation. This attribute comes into play when the KPDS_*_SCALING attribute is set to KNORMALIZE.

Map Segment Attributes		
Attribute and Default	Legal Values	Definition
		Persistence: transient
KPDS_MAP_NORM_MIN double norm_min	< norm_max	This attribute specifies the minimum to be used when normalizing data values. This attribute is used in conjunction with the KPDS_*_NORM_MAX attribute, respectively, to determine the bounds of a normalization operation. This attribute comes into play when the KPDS_*_SCALING attribute is set to KNORMALIZE.
		Persistence: transient
KPDS_MAP_OFFSET int mw 0 mh 0 md 0 mt 0 me 0	+/- int	These attribute values specify the offset into the data position for all primitives. Offset values beyond the boundaries of the data are valid.
		Persistence: transient
KPDS_MAP_OPTIMAL_REGION_SIZE int region_width region_height region_depth region_time region_elements number_of_regions	> 0	This attribute will return the size of a region of data and the number of such regions that is most efficient to process in terms of performance and memory use. The KPDS_*_REGION_SIZE attribute controls the size and dictates the number of regions, which will always be rounded up. See KPDS_*_REGION_SIZE for more information.
		Persistence: transient
KPDS_MAP_PAD_VALUE double real 0.0 imag 0.0		This attribute specifies the real (and imaginary) values of the pad data if the KPDS_*_INTERPOLATE attribute is set to KPAD, respectively. The double values must be specified, whether the data is real or complex. The pad values will internally be converted from double to the appropriate data type. The default pad value for <i>location</i> , <i>map</i> , <i>time</i> and <i>value</i> is 0. The default pad value for the <i>mask</i> is 1 because when padding data, the padded portion of the <i>value</i> data should initially be considered valid.
		Persistence: transient

Map Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_MAP_PLANE_INFO int map_plane_width map_plane_height num_planes		This attribute will return the size of a plane of data in points and the number of planes in the dataset. The plane size will be the size of the width and height axes and the number will be the product of the sizes of the other axes. Persistence: transient
KPDS_MAP_POSITION int mw 0 mh 0 md 0 mt 0 me 0		The position attribute specifies four indices to locate a specific PRIMITIVE in the <i>location</i> data. <i>w</i> is <i>width</i> , <i>h</i> is <i>height</i> , <i>d</i> is <i>depth</i> and <i>n</i> is <i>dimension</i> . Persistence: transient
KPDS_MAP_REGION_INFO int region_width region_height region_depth region_time region_elements num_regions		This attribute will return the size of a region of data in points and the number of regions in the data. The KPDS_*_REGION_SIZE attribute controls the size and dictates the number of regions, which will always be rounded up. See KPDS_*_REGION_SIZE for more information. Persistence: transient
KPDS_MAP_REGION_SIZE int region_width 1 region_height 1 region_depth 1 region_time 1 region_elements 1	> 0	These attribute values specify the size of the region being processed. If the region size for one or more dimensions is not an even multiple of the data size, then the pad value will be returned by <code>kpds_get_data</code> for all data outside of the data space, which is set by <code>KPDS_*_PAD_VALUE</code> . On a <code>kpds_put_data</code> call, data points outside of the data space will be truncated. E.g. if the object width is 512 and the region width is 200, then getting the first two regions will return the data for those regions. The next get will return the remaining 112 points from the data in the width direction with the remaining 88 points set to the pad value. On a put for this same setup, the first two puts will place full regions into the data object, but the last put will place only the first 112 points into the data object in the width direction and the last 88 points are truncated. Persistence: transient

Map Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_MAP_SCALE_FACTOR double scale_factor	1.0	This attribute specifies the scaling factor to be used when scaling data values. This attribute comes into play when the KPDS_*_SCALING attribute is set to KSCALE, respectively. Persistence: transient
KPDS_MAP_SCALE_OFFSET double offset_real 0.0 offset_imaginary 0.0		This attribute specifies the scaling offset to be used when scaling data values. This attribute comes into play when the KPDS_*_SCALING attribute is set to KSCALE, respectively. Persistence: transient
KPDS_MAP_SCALING int scaling KNONE	KNONE KNORMALIZE KSCALE	This attribute specifies whether scaling or normalization should be performed. If KSCALE is specified for the <i>value</i> data, values will be scaled, according to the KPDS_VALUE_SCALE_FACTOR and KPDS_VALUE_SCALE_OFFSET attributes. If KNORMALIZE is specified for the <i>value</i> data, values will be normalized using the KPDS_VALUE_NORM_MIN and KPDS_VALUE_NORM_MAX attributes. If this attribute is set to KNONE for the value data, values will not be scaled or normalized. The same is true for the <i>map</i> and <i>mask</i> data. They will use their respective scale factor & offset and normalize minimum & maximum attributes. Persistence: transient
KPDS_MAP_SIZE int map_width map_height map_depth map_time map_elements	> 0	This attribute specifies the size of the dimensions <i>width</i> , <i>height</i> , <i>depth</i> , <i>elements</i> , <i>time</i> , <i>location dimension</i> , <i>map width</i> , <i>map height</i> , <i>map elements</i> , <i>map depth</i> <i>map time</i> . When the application programmer specifies a size larger than the actual size of stored data, the get functions will sub-sampled, clipped, padded or duplicated the data to present the program with the requested amount, see the attribute KPDS_VALUE_INTERPOLATE for more details. The put functions store exactly the size that the physical attributes will allow even if the amount of data "put" (set by the presentation attributes) is different. This attribute must be set for objects created via <code>kpds_create_object</code> or output objects else get/put data calls will fail. The size of the <i>mask</i> and <i>value</i> data is identical. The <i>time</i> size is shared between the <i>time</i> , <i>mask</i> and <i>value</i> data. The <i>width</i> , <i>height</i> <i>depth</i> are shared between the <i>location</i> , <i>mask</i> <i>value</i> data.

Map Segment Attributes		
Attribute and Default	Legal Values	Definition
		Persistence: stored
KPDS_MAP_VECTOR_INFO int vector_length num_vectors		This attribute will return the number of points in a vector of data and the number of vectors in the dataset. The vector definition for the data primitive. For KPDS_VALUE_VECTOR_INFO, the size is the size of the element vector and the number is the product of the remaining dimensions.
		Persistence: transient
KPDS_MAP_VOLUME_INFO int map_width map_height map_depth num_volumes		This attribute will return the size of a volume of data in points and the number of volumes in the dataset. The volume size will be the size of the width, height, and depth axes. The number volumes will be the product of the sizes of the remaining axes.
		Persistence: transient

F.5. Location Segment Attributes

Location Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_LOCATION_BEGIN double w 0.0 h 0.0 d 0.0		This attribute represents an explicit begin marker point for the polymorphic data model. This begin point maps to the implicit origin of the data model. This attribute can only be set if uniform <i>location</i> data has been explicitly created with a <code>kpds_create_location</code> call. See also: <code>KPDS_LOCATION_END</code> . Persistence: transient
KPDS_LOCATION_COMPLEX_CONVERT int convert KREAL	KIMAGINARY KMAGNITUDE KPHASE KREAL KMAGSQ KMAGSQP1 KLOGMAG KLOGMAGP1	This attribute specifies how complex data should be converted. If it is converted to a "lower" data type, this attribute specifies how to down-convert the data. For example if the data is actually complex, but the presentation attribute is byte, the complex data would first be converted to the representation defined by this attribute, and then converted to byte. If the data is being converted from a "lower" data type to a complex data type, this attribute defines how the data should be interpreted — as the real or imaginary component of the complex pair. <code>KPHASE</code> and <code>KMAGNITUDE</code> are invalid values for up converting to complex, and will result in an error. Persistence: transient
KPDS_LOCATION_DATA_TYPE int datatype	KBIT KBYTE KUBYTE KSHORT KUSHORT KINT KUINT KLONG KULONG KPLOAT KDOUBLE KCOMPLEX KDComplex	This attribute is used to get or set the data type, or numerical representation of the data. This data type will be the presentation data type, not necessarily the physical data type. See the <code>KPDS_COUPLING</code> attribute for more information on how to control the presentation and physical data types. When the application programmer specifies a presentation data type that is different than the actual data type of the stored data, the <code>get kpds_get_data</code> function will convert the data to return the requested data type. Likewise, the <code>kpds_put_data</code> function expects data that is in the data type specified by this attribute to the output object, and if the data being "put" is of a different type, it will be converted. This attribute must be set for objects created via <code>kpds_create_object</code> or output objects that are opened with <code>kpds_open_output_object</code> or <code>kpds_open_object</code> , or else the get and put data calls will fail. Persistence: stored

Location Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_LOCATION_END double w 0.0 h 0.0 d 0.0		This attribute represents an explicit end marker point for the polymorphic data model. This end point maps to the implicit extent of the data model. This attribute can only be set if uniform <i>location</i> data has been explicitly created with a <code>kpds_create_location</code> call. See also: <code>KPDS_LOCATION_BEGIN</code> . Persistence: transient
KPDS_LOCATION_GRID int grid_type	KNONE KUNIFORM KRECTILINEAR KCURVILINEAR	The <i>location</i> grid attribute specifies the grid type to use when creating the <i>location</i> data. This attribute should be set before a <code>kpds_create_location</code> operation. By default, if the <i>location</i> data type is <code>KNONE</code> and <i>location</i> data is created, a curvilinear <i>location</i> grid will be created and this attribute will be set to <code>KCURVILINEAR</code> . Persistence: permanent
KPDS_LOCATION_INCREMENT_SIZE int width height depth num_volumes		This attribute is used to alter how the auto-increment state machine behaves. Normally, the size of the data set is used to dictate how position auto-advances from one position to the next, based on the primitive being accessed. This attribute, if set, will be used instead of the size of the data set for controlling auto-advancement. One place where such functionality is useful is when a smaller data set is being inserted into a larger one. The larger destination data set's <code>INCREMENT_SIZE</code> attribute can be set to the size of the smaller source data set so that the auto-advancement stays synchronized across all dimensions. Persistence: transient

Location Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_LOCATION_INTERPOLATE int interpolate KPAD	KNONE KPAD KZERO_ORDER KWRAP	<p>This attribute specifies how the data should be presented if the application program requests a size different from what is physically stored. If the size requested is larger than the physical size and the interpolation requested is KPAD the pad value will be returned for all points outside of the physical size. If the size requested is smaller than the physical size and the interpolation requested is KPAD the returned data is clipped to the size requested. If the size requested is larger than the physical size and the interpolation requested is KZERO_ORDER the data is duplicated. If the size requested is smaller than the physical size and the interpolation requested is KZERO_ORDER the data is sub-sampled. If the interpolation requested is KWRAP then the size change will be resolved by duplicating the data set. If KWRAP is set, then out-of-bounds data accesses will also be filled with duplicated data. If the interpolate attribute is set to KNONE, an error will be returned if the program requests a size different from what is physically stored.</p> <p>Persistence: transient</p>
KPDS_LOCATION_LINE_INFO int line_size num_lines		<p>This attribute will return the number of points in a line and the number of lines in the dataset. The line size will be the size of the width axis and the of lines number will be the product of the other axes' sizes.</p> <p>Persistence: transient</p>
KPDS_LOCATION_NORM_MAX double norm_max	> norm_min	<p>This attribute specifies the maximum to be used when normalizing data values. This attribute is used in conjunction with the KPDS_*_NORM_MIN attribute, respectively, to determine the bounds of a normalization operation. This attribute comes into play when the KPDS_*_SCALING attribute is set to KNORMALIZE.</p> <p>Persistence: transient</p>
KPDS_LOCATION_NORM_MIN double norm_min unknown	< norm_max	<p>This attribute specifies the minimum to be used when normalizing data values. This attribute is used in conjunction with the KPDS_*_NORM_MAX attribute, respectively, to determine the bounds of a normalization operation. This attribute comes into play when the KPDS_*_SCALING attribute is set to KNORMALIZE.</p> <p>Persistence: transient</p>

Location Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_LOCATION_OFFSET int w 0 h 0 d 0 n 0	+/- int	These attribute values specify the offset into the data position for all primitives. Offset values beyond the boundaries of the data are valid. Persistence: transient
KPDS_LOCATION_OPTIMAL_REGION_SIZE int w h d dim num_regions	> 0	This attribute will return the size of a region of data and the number of such regions that is most efficient to process in terms of performance and memory use. The KPDS_*_REGION_SIZE attribute controls the size and dictates the number of regions, which will always be rounded up. See KPDS_*_REGION_SIZE for more information. Persistence: transient
KPDS_LOCATION_PAD_VALUE double real 0.0 imag 0.0		This attribute specifies the real (and imaginary) values of the pad data if the KPDS_*_INTERPOLATE attribute is set to KPAD, respectively. The double values must be specified, whether the data is real or complex. The pad values will internally be converted from double to the appropriate data type. The default pad value for <i>location</i> , <i>map</i> , <i>time</i> and <i>value</i> is 0. The default pad value for the <i>mask</i> is 1 because when padding data, the padded portion of the <i>value</i> data should initially be considered valid. Persistence: transient
KPDS_LOCATION_PLANE_INFO int w h num_planes		This attribute will return the size of a plane of data in points and the number of planes in the dataset. The plane size will be the size of the width and height axes and the number will be the product of the sizes of the other axes. Persistence: transient

Location Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_LOCATION_POSITION int w 0 h 0 d 0 n 0	+/- int	The position attribute specifies four indices to locate a specific PRIMITIVE in the <i>location</i> data. <i>w</i> is <i>width</i> , <i>h</i> is <i>height</i> , <i>d</i> is <i>depth</i> and <i>n</i> is <i>dimension</i> . Persistence: transient
KPDS_LOCATION_REGION_INFO int w h d dim num_regions		This attribute will return the size of a region of data in points and the number of regions in the data. The KPDS_*_REGION_SIZE attribute controls the size and dictates the number of regions, which will always be rounded up. See KPDS_*_REGION_SIZE for more information. Persistence: transient
KPDS_LOCATION_REGION_SIZE int w 1 h 1 d 1 dim 1	> 0	These attribute values specify the size of the region being processed. If the region size for one or more dimensions is not an even multiple of the data size, then the pad value will be returned by <code>kpds_get_data</code> for all data outside of the data space, which is set by <code>KPDS_*_PAD_VALUE</code> . On a <code>kpds_put_data</code> call, data points outside of the data space will be truncated. E.g. if the object width is 512 and the region width is 200, then getting the first two regions will return the data for those regions. The next get will return the remaining 112 points from the data in the width direction with the remaining 88 points set to the pad value. On a put for this same setup, the first two puts will place full regions into the data object, but the last put will place only the first 112 points into the data object in the width direction and the last 88 points are truncated. Persistence: transient
KPDS_LOCATION_SCALE_FACTOR double scale_factor 1.0		This attribute specifies the scaling factor to be used when scaling data values. This attribute comes into play when the <code>KPDS_*_SCALING</code> attribute is set to <code>KSCALE</code> , respectively. Persistence: transient

Location Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_LOCATION_SCALE_OFFSET double real 0.0 offset 0.0		This attribute specifies the scaling offset to be used when scaling data values. This attribute comes into play when the KPDS_*_SCALING attribute is set to KSCALE, respectively. Persistence: transient
KPDS_LOCATION_SCALING int scaling KNONE	KNONE KNORMALIZE KSCALE	This attribute specifies whether scaling or normalization should be performed. If KSCALE is specified for the <i>value</i> data, values will be scaled, according to the KPDS_VALUE_SCALE_FACTOR and KPDS_VALUE_SCALE_OFFSET attributes. If KNORMALIZE is specified for the <i>value</i> data, values will be normalized using the KPDS_VALUE_NORM_MIN and KPDS_VALUE_NORM_MAX attributes. If this attribute is set to KNONE for the value data, values will not be scaled or normalized. The same is true for the <i>map</i> and <i>mask</i> data. They will use their respective scale factor & offset and normalize minimum & maximum attributes. Persistence: transient
KPDS_LOCATION_SIZE int w h d dim	> 0	This attribute specifies the size of the dimensions <i>width</i> , <i>height</i> , <i>depth</i> , <i>elements</i> , <i>time</i> , <i>location dimension</i> , <i>map width</i> , <i>map height</i> , <i>map elements</i> , <i>map depth</i> <i>map time</i> . When the application programmer specifies a size larger than the actual size of stored data, the get functions will sub-sampled, clipped, padded or duplicated the data to present the program with the requested amount, see the attribute KPDS_VALUE_INTERPOLATE for more details. The put functions store exactly the size that the physical attributes will allow even if the amount of data "put" (set by the presentation attributes) is different. This attribute must be set for objects created via <code>kpds_create_object</code> or output objects else get/put data calls will fail. The size of the <i>mask</i> and <i>value</i> data is identical. The <i>time</i> size is shared between the <i>time</i> , <i>mask</i> and <i>value</i> data. The <i>width</i> , <i>height</i> <i>depth</i> are shared between the <i>location</i> , <i>mask</i> <i>value</i> data. Persistence: stored
KPDS_LOCATION_VECTOR_INFO int vector_length num_vectors		This attribute will return the number of points in a vector of data and the number of vectors in the dataset. The vector definition for the data primitive. For KPDS_VALUE_VECTOR_INFO, the size is the size of the element vector and the number is the product of the remaining dimensions.

Location Segment Attributes		
Attribute and Default	Legal Values	Definition
		Persistence: transient
KPDS_LOCATION_VOLUME_INFO int w h d num_volumes		This attribute will return the size of a volume of data in points and the number of volumes in the dataset. The volume size will be the size of the width, height, and depth axes. The number volumes will be the product of the sizes of the remaining axes.
		Persistence: transient

F.6. Time Segment Attributes

Time Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_TIME_COMPLEX_CONVERT int convert KREAL	KIMAGINARY KMAGNITUDE KPHASE KREAL KMAGSQ KMAGSQP1 KLOGMAG KLOGMAGP1	This attribute specifies how complex data should be converted. If it is converted to a "lower" data type, this attribute specifies how to down-convert the data. For example if the data is actually complex, but the presentation attribute is byte, the complex data would first be converted to the representation defined by this attribute, and then converted to byte. If the data is being converted from a "lower" data type to a complex data type, this attribute defines how the data should be interpreted — as the real or imaginary component of the complex pair. KPHASE and KMAGNITUDE are invalid values for up converting to complex, and will result in an error.
		Persistence: transient

Time Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_TIME_DATA_TYPE int datatype	KBIT KBYTE KUBYTE KSHORT KUSHORT KINT KUINT KLONG KULONG KFLOAT KDOUBLE KCOMPLEX KDCOMPLEX	<p>This attribute is used to get or set the data type, or numerical representation of the data. This data type will be the presentation data type, not necessarily the physical data type. See the <code>KPDS_COUPLING</code> attribute for more information on how to control the presentation and physical data types. When the application programmer specifies a presentation data type that is different than the actual data type of the stored data, the <code>get kpds_get_data</code> function will convert the data to return the requested data type. Likewise, the <code>kpds_put_data</code> function expects data that is in the data type specified by this attribute to the output object, and if the data being "put" is of a different type, it will be converted. This attribute must be set for objects created via <code>kpds_create_object</code> or output objects that are opened with <code>kpds_open_output_object</code> or <code>kpds_open_object</code>, or else the get and put data calls will fail.</p> <p>Persistence: stored</p>
KPDS_TIME_INCREMENT_SIZE int width height depth num_volumes		<p>This attribute is used to alter how the auto-increment state machine behaves. Normally, the size of the data set is used to dictate how position auto-advances from one position to the next, based on the primitive being accessed. This attribute, if set, will be used instead of the size of the data set for controlling auto-advancement. One place where such functionality is useful is when a smaller data set is being inserted into a larger one. The larger destination data set's <code>INCREMENT_SIZE</code> attribute can be set to the size of the smaller source data set so that the auto-advancement stays synchronized across all dimensions.</p> <p>Persistence: transient</p>

Time Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_TIME_INTERPOLATE int interpolate KPAD	KNONE KPAD KZERO_ORDER KWRAP	<p>This attribute specifies how the data should be presented if the application program requests a size different from what is physically stored. If the size requested is larger than the physical size and the interpolation requested is KPAD the pad value will be returned for all points outside of the physical size. If the size requested is smaller than the physical size and the interpolation requested is KPAD the returned data is clipped to the size requested. If the size requested is larger than the physical size and the interpolation requested is KZERO_ORDER the data is duplicated. If the size requested is smaller than the physical size and the interpolation requested is KZERO_ORDER the data is sub-sampled. If the interpolation requested is KWRAP then the size change will be resolved by duplicating the data set. If KWRAP is set, then out-of-bounds data accesses will also be filled with duplicated data. If the interpolate attribute is set to KNONE, an error will be returned if the program requests a size different from what is physically stored.</p> <p>Persistence: transient</p>
KPDS_TIME_NORM_MAX double norm_max	> norm_min	<p>This attribute specifies the maximum to be used when normalizing data values. This attribute is used in conjunction with the KPDS_*_NORM_MIN attribute, respectively, to determine the bounds of a normalization operation. This attribute comes into play when the KPDS_*_SCALING attribute is set to KNORMALIZE.</p> <p>Persistence: transient</p>
KPDS_TIME_NORM_MIN double norm_min	< norm_max	<p>This attribute specifies the minimum to be used when normalizing data values. This attribute is used in conjunction with the KPDS_*_NORM_MAX attribute, respectively, to determine the bounds of a normalization operation. This attribute comes into play when the KPDS_*_SCALING attribute is set to KNORMALIZE.</p> <p>Persistence: transient</p>
KPDS_TIME_OFFSET int offset_time 0	+/- int	<p>These attribute values specify the offset into the data position for all primitives. Offset values beyond the boundaries of the data are valid.</p> <p>Persistence: transient</p>

Time Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_TIME_OPTIMAL_REGION_SIZE int t num		This attribute will return the size of a region of data and the number of such regions that is most efficient to process in terms of performance and memory use. The KPDS_*_REGION_SIZE attribute controls the size and dictates the number of regions, which will always be rounded up. See KPDS_*_REGION_SIZE for more information. Persistence: transient
KPDS_TIME_PAD_VALUE double real_value 0.0 imag_value 0.0		This attribute specifies the real (and imaginary) values of the pad data if the KPDS_*_INTERPOLATE attribute is set to KPAD, respectively. The double values must be specified, whether the data is real or complex. The pad values will internally be converted from double to the appropriate data type. The default pad value for <i>location</i> , <i>map</i> , <i>time</i> and <i>value</i> is 0. The default pad value for the <i>mask</i> is 1 because when padding data, the padded portion of the <i>value</i> data should initially be considered valid. Persistence: transient
KPDS_TIME_POSITION int t 0	+/- int	This attribute specifies the real (and imaginary) values of the pad data if the KPDS_*_INTERPOLATE attribute is set to KPAD, respectively. The double values must be specified, whether the data is real or complex. The pad values will internally be converted from double to the appropriate data type. The default pad value for <i>location</i> , <i>map</i> , <i>time</i> and <i>value</i> is 0. The default pad value for the <i>mask</i> is 1 because when padding data, the padded portion of the <i>value</i> data should initially be considered valid. Persistence: transient
KPDS_TIME_REGION_INFO int region_time num_regions		This attribute will return the size of a region of data in points and the number of regions in the data. The KPDS_*_REGION_SIZE attribute controls the size and dictates the number of regions, which will always be rounded up. See KPDS_*_REGION_SIZE for more information. Persistence: transient

Time Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_TIME_REGION_SIZE int region_time 1	> 0	<p>These attribute values specify the size of the region being processed. If the region size for one or more dimensions is not an even multiple of the data size, then the pad value will be returned by <code>kpds_get_data</code> for all data outside of the data space, which is set by <code>KPDS_*_PAD_VALUE</code>. On a <code>kpds_put_data</code> call, data points outside of the data space will be truncated. E.g. if the object width is 512 and the region width is 200, then getting the first two regions will return the data for those regions. The next get will return the remaining 112 points from the data in the width direction with the remaining 88 points set to the pad value. On a put for this same setup, the first two puts will place full regions into the data object, but the last put will place only the first 112 points into the data object in the width direction and the last 88 points are truncated.</p> <p>Persistence: transient</p>
KPDS_TIME_SCALE_FACTOR double scale_factor 1.0		<p>This attribute specifies the scaling factor to be used when scaling data values. This attribute comes into play when the <code>KPDS_*_SCALING</code> attribute is set to <code>KSCALE</code>, respectively.</p> <p>Persistence: transient</p>
KPDS_TIME_SCALE_OFFSET double offset_real 0.0 offset_imaginary 0.0		<p>This attribute specifies the scaling offset to be used when scaling data values. This attribute comes into play when the <code>KPDS_*_SCALING</code> attribute is set to <code>KSCALE</code>, respectively.</p> <p>Persistence: transient</p>
KPDS_TIME_SCALING int scaling KNONE	KNONE KNORMALIZE KSCALE	<p>This attribute specifies whether scaling or normalization should be performed.</p> <p>If <code>KSCALE</code> is specified for the <i>value</i> data, values will be scaled, according to the <code>KPDS_VALUE_SCALE_FACTOR</code> and <code>KPDS_VALUE_SCALE_OFFSET</code> attributes. If <code>KNORMALIZE</code> is specified for the <i>value</i> data, values will be normalized using the <code>KPDS_VALUE_NORM_MIN</code> and <code>KPDS_VALUE_NORM_MAX</code> attributes. If this attribute is set to <code>KNONE</code> for the value data, values will not be scaled or normalized. The same is true for the <i>map</i> and <i>mask</i> data. They will use their respective scale factor & offset and normalize minimum & maximum attributes.</p> <p>Persistence: transient</p>

Time Segment Attributes		
Attribute and Default	Legal Values	Definition
KPDS_TIME_SIZE int t	> 0	This attribute specifies the size of the dimensions <i>width</i> , <i>height</i> , <i>depth</i> , <i>elements</i> , <i>time</i> , <i>location dimension</i> , <i>map width</i> , <i>map height</i> , <i>map elements</i> , <i>map depth</i> <i>map time</i> . When the application programmer specifies a size larger than the actual size of stored data, the get functions will sub-sampled, clipped, padded or duplicated the data to present the program with the requested amount, see the attribute KPDS_VALUE_INTERPOLATE for more details. The put functions store exactly the size that the physical attributes will allow even if the amount of data "put" (set by the presentation attributes) is different. This attribute must be set for objects created via <code>kpds_create_object</code> or output objects else get/put data calls will fail. The size of the <i>mask</i> and <i>value</i> data is identical. The <i>time</i> size is shared between the <i>time</i> , <i>mask</i> and <i>value</i> data. The <i>width</i> , <i>height</i> <i>depth</i> are shared between the <i>location</i> , <i>mask</i> <i>value</i> data.
		Persistence: stored

G. Functions Provided by Polymorphic Data Services

The API functions that are provided by Polymorphic Data Services are described below. They are organized into sections according to their classification as management, data, or attribute operators.

Note: The multiple attribute functions `kpds_get_attributes()`, `kpds_match_attributes()`, and `kpds_set_attributes()` require a NULL at the end of the variable argument list to indicate the end of the list.

G.1. Object Management

- `kpds_open_input_object()` - open an input object for reading
- `kpds_open_output_object()` - open an output object for writing
- `kpds_create_object()` - create a temporary data object.
- `kpds_create_object_attr()` - create an attribute associated the data object.
- `kpds_destroy_object_attr()` - destroy an attribute associated with the data object.
- `kpds_open_object()` - create an object associated with an input or output transport.
- `kpds_close_object()` - close an open data object.
- `kpds_reference_object()` - create a reference of a data object.
- `kpds_copy_object()` - copy all data and attributes from one object to another.
- `kpds_copy_object_attr()` - copy all presentation attributes from one data object to another.
- `kpds_copy_object_data()` - copy all data from one object to another object.
- `kpds_copy_remaining_data()` - copy remaining data from source to destination
- `kpds_sync_object()` - synchronize physical and presentation layers of a data object.

G.1.1. `kpds_open_input_object()` — *open an input object for reading*

Synopsis

```
kobject kpds_open_input_object(  
  
    char *name)
```

Input Arguments

`name`
a string that contains the path name of a file or transport that will be associated with the object.

Returns

a `kobject` on success, `KOBJECT_INVALID` on failure.

Description

This function is a simplified interface to the `kpds_open_object` function. It differs from `kpds_open_object` in that it assumes that the object is read-only and its transport has permanence. If a permanent file is not desired (i.e. the object is going to be used as temporary storage, and will not be used by any other process) then the `kpds_create_object` function call should be used instead.

The argument to this function is the transport or file name. This argument indicates the name of the transport that is associated with the object. The transport name can be any legal VisiQuest transport description. While this will usually be a regular UNIX file name, it is also possible to specify such things as shared memory pointers, memory map pointers, sockets, streams, and even transports on other machines. For more information on the syntax of a VisiQuest transport name, refer to the online man page for the VisiQuest function `kopen`. For more information about opening an object, see `kpds_open_object`.

The presentation index order will be set to `KWIDTH`, `KHEIGHT`, `KDEPTH`, `KTIME` and `KELEMENTS` for the value and mask data, to `KMAP_WIDTH`, `KMAP_HEIGHT`, `KMAP_ELEMENTS`, `KMAP_SPATIAL` and `KMAP_TIME` for the map data and `KWIDTH`, `KHEIGHT`, `KDEPTH` and `KDIMENSION` for the location data. The only way to get the index order to reflect the stored index order of the data is to call `kpds_sync_object`. See the man page for `kpds_sync_object` for more information.

Because this function opens an input object, the `KPDS_COUPLING` attribute is set to `KPDS_UNCOUPLED`.

This function is equivalent to:

```
kpds_open_object(name, KOBJ_READ)
```

G.1.2. `kpds_open_output_object()` — *open an output object for writing*

Synopsis

```
kobject kpds_open_output_object(  
  
    char *name)
```

Input Arguments

`name`
a string that contains the path name of a file or transport that will be associated with the object.

Returns

a kobject on success, `KOBJECT_INVALID` on failure.

Description

This function is a simplified interface to the `kpds_open_object` function. It differs from `kpds_open_object` in that it assumes that the object is write-only and its transport has permanence. If a permanent file is not desired (i.e. the object is going to be used as temporary storage, and will not be used by any other process) then the `kpds_create_object` function call should be used instead.

The argument to this function is the transport or file name. This argument indicates the name of the transport that is associated with the object. The transport name can be any legal khoros transport description. While this will usually be a regular UNIX file name, it is also possible to specify such things as shared memory pointers, memory map pointers, sockets, streams, and even transports on other machines. For more information on the syntax of a VisiQuest transport name, refer to the online man page for the VisiQuest function `kopen`. For more information about opening objects, see `kpds_open_object`.

The default index order will be set to `KWIDTH`, `KHEIGHT`, `KDEPTH`, `KTIME` and `KELEMENTS` for the value and mask data, to `KMAP_WIDTH`, `KMAP_HEIGHT`, `KMAP_ELEMENTS`, `KMAP_SPATIAL` and `KMAP_TIME` for the map data and `KWIDTH`, `KHEIGHT`, `KDEPTH` and `KDIMENSION` for the location data.

This function is equivalent to:

```
kpds_open_object(name, KOBJ_WRITE)
```

G.1.3. `kpds_create_object()` — *create a temporary data object.*

Synopsis

```
kobject kpds_create_object(void)
```

Returns

kobject on success, KOBJECT_INVALID upon failure

Description

This function is used to instantiate a data object (kobject) when it will only be used for temporary storage of information. If you are intending to process an object that already exists as a file or transport (input), or you are planning on saving the kobject to a file or transport (output), then the appropriate routines to use are `kpds_open_input_object`, `kpds_open_output_object`, or `kpds_open_object`.

This function creates an instance of a data object that will have associated with it a temporary transport that will be used for buffering large amounts of data. This temporary transport will be automatically removed when the process terminates. There is no way to rename the temporary file or replace the temporary file with a permanent one.

The `kpds_create_object` function call creates what is essentially a "blank" object. That is, the object will initially have no data and all attributes will be initialized to default values or to an initialized state. The default values for attributes are described in the Khoros 2.0 Programming Services Volume 2 Manual.

An object that is created with this function call behaves similarly to an output object that is created with the `kpds_open_output_object` function call, i.e. the object initially has no data or attributes. Thus, it is necessary to create the location, map, mask, time and/or value data and initialize attributes such as size and datatype prior to using the object.

G.1.4. `kpds_create_object_attr()` — *create an attribute associated the data object.*

Synopsis

```
int kpds_create_object_attr(  
  
    kobject object,  
    char    *attribute,  
    int     num_args,  
    int     arg_size,  
    int     data_type,  
    int     permanent,  
    int     shared)
```


Input Arguments

`object`

the object on which to instantiate the new attribute.

`attribute`

attribute identifier string. This identifier must be unique for the given segment.

`numargs` - number of arguments in the attribute; must be > 0;

`argsize` - number of units of the data type for each attribute argument must be > 0;

`datatype` - data type of the attribute can be KBIT, KUBYTE, KBYTE, KUSHORT, KSHORT, KUINT, KINT, KULONG, KLONG, KFLOAT, KDOUBLE, KCOMPLEX, KDCOMPLEX, or KSTRING

`permanent`

TRUE if attribute is permanent.

`shared`

TRUE if attribute is shared

Returns

TRUE on success, FALSE otherwise

Description

The purpose of this routine is to provide the programmer with a mechanism for creating attributes that are specific to the program being written. The attribute created will be associated with the data object itself and not with the Value, Mask, Map, Location or Time data.

The second argument to this function is the name of the attribute to be created specified as a string. If the attribute specified already exists then this function will return FALSE.

The datatype argument indicates the data type of all the elements associated with the attribute. It take on any of the following values: KUBYTE, KBYTE, KUSHORT, KSHORT, KUINT, KINT, KULONG, KLONG, KFLOAT, KDOUBLE, KCOMPLEX, or KDCOMPLEX. Any other value is considered illegal. Passing this argument with an illegal value will force this function to return FALSE.

The numargs argument indicates how many of elements of the type "datatype" will be contained in the attribute. This information is necessary so that Polymorphic Services can allocate sufficient memory to store the attribute. A negative or 0 value passed in will force this function to return FALSE.

The argsize argument indicates the size of each element, (i.e. you could be passing in a pointer to an array of integers or several arrays if num greater than 1). This information is necessary so that Polymorphic Services can allocate sufficient memory to store the attribute. This argument specifies the length of the array that needs to be allocated.

The last argument, permanent, indicates whether the attribute should be saved when the data object is

closed and being stored in a permanent transport. It should be set to TRUE if the attribute is permanent and FALSE if the attribute is transient.

For example, calling this function as follows:

```
kpds_create_object_attr(object, "Nose", 1, 10, KFLOAT, FALSE);
```

will create an attribute called "Nose" that contains 10 floats that are accessed as a single array. A call to `kpds_get_attribute` might look like this:

```
float *array;  
kpds_get_attribute(object, "Nose", &array);
```

Restrictions

The length variable indicates the length of all the attribute elements. If different size lengths are desired, different attributes must be created.

G.1.5. `kpds_destroy_object_attr()` — *destroy an attribute associated with the data object.*

Synopsis

```
int kpds_destroy_object_attr(  
  
    kobject object,  
    char *attribute)
```

Input Arguments

`object`
the object that contains the attribute.
`attribute`
the name of the attribute to destroy.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

The purpose of this routine is to provide the programmer with a mechanism for deleting attributes that were previously created with a call to `kpds_create_object_attr`.

The name argument specifies the name of the attribute to be destroyed. This function will fail if the attribute is one of the predefined KPDS attributes or if the specified attribute does not exist.

G.1.6. kpds_open_object() — *create an object associated with an input or output transport.*

Synopsis

```
kobject kpds_open_object(  
    char *name,  
    int  flags)
```

Input Arguments

name

a string that contains the path name of a file or transport that will be associated with the object.

flags

how the object is to be opened. A bitwise OR of KOBJ_READ, KOBJ_WRITE, KOBJ_RAW as described above.

Returns

kobject on success, KOBJECT_INVALID upon failure

Description

This function is used to instantiate a data object (kobject) that is associated with a permanent file or transport. If a permanent file is not desired (i.e. the object is going to be used as temporary storage, and will not be used by any other process) then the kpds_create_object function call should be used instead.

The first argument to this function is the transport or file name. This argument indicates the name of the transport that is associated with the object. The transport name can be any legal khoros transport description. While this will usually be a regular UNIX file name, it is also possible to specify such things as shared memory pointers, memory map pointers, sockets, streams, and even transports on other machines. For more information on the syntax of a VisiQuest transport name, refer to the online man page for the VisiQuest function kopen.

The second argument to the kpds_open_object function call, flags, is used to provide polymorphic services with specific information about how the object is going to be manipulated. The flags argument is analogous to kopen's flags argument. The flags argument is constructed by bitwise OR'ing predefined values from the following list:

KOBJ_READ - Open an existing file or transport for reading (input). By using this flag, you are indicating that the file or transport exists and that it contains valid data. If it does not exist, or the data is not recognized, then an error message will be generated and this function will return KOBJECT_INVALID.

- KOBJ_WRITE - Open a file or transport for writing (output). By using this flag, you are indicating that any data that you write to the object will be stored in the file or transport specified.
- KOBJ_RAW - When an object is opened, data services usually attempts to recognize the file format by examining the first part of the file. By setting this value, you will bypass this operation, forcing the file to be read as raw unformatted data.

These flags can be combined arbitrarily to determine how the file or transport is opened and subsequently manipulated. This is done by bitwise OR'ing any combination of these options. For example KOBJ_READ | KOBJ_WRITE will result in a read/write file object. This implies that the file already exists and will be read from using `kpds_get_data` and written to using `kpds_put_data`. When `kpds_close` is called, the changes that are a result of calls to `kpds_put_data` will be stored to the file or transport.

However, if you intend to open an output object, but you need to occasionally read data from it that you have already written, it is not necessary to specify KOBJ_READ (in fact, doing so may result result in an error if the file or transport does not already exist).

Likewise, it is possible to call `kpds_put_data` on an input object (one which was opened without the KOBJ_WRITE flag). If this is done, then subsequent calls to `kpds_get_data` on a region that has been written to will contain the new data. However, the file or transport that is associated with this input object will not be changed. Thus, the KOBJ_READ and KOBJ_WRITE flags only indicate what operations are allowed on the permanent file or transport that is associated with the object, not what operations are allowable on the object itself.

If KOBJ_READ is specified, then the Data Services will attempt to recognize the file format automatically. If it fails, then this function will return `KOBJECT_INVALID`, indicating that it was unable to open the object, unless the KOBJ_RAW flag was also specified, in which case, it will assume that the input file is simply raw data. The structured file formats that are currently recognized are VIFF (The Khoros 2.0 standard file format), Viff (The Khoros 1.0 standard file format, which was referred to as VIFF in Khoros 1.0), Pnm (Portable Any Map, which includes PBM, PGM, and PNM), and Sun Raster.

The default index order will be set to `KWIDTH`, `KHEIGHT`, `KDEPTH`, `KTIME` and `KELEMENTS` for the value and mask data, to `KMAP_WIDTH`, `KMAP_HEIGHT`, `KMAP_ELEMENTS`, `KMAP_DEPTH` and `KMAP_TIME` for the map data and `KWIDTH`, `KHEIGHT`, `KDEPTH` and `KDIMENSION` for the location data. The only way to get the index order to reflect the stored index order of the data is to call `kpds_sync_object`. See the man page for `kpds_sync_object` for more information.

Restrictions

The KOBJ_RAW flag will have unpredictable results if it is combined with the KOBJ_WRITE flag. This limitation will be removed in a later release of the Khoros 2.0 system.

G.1.7. `kpds_close_object()` — *close an open data object.*

Synopsis

```
int kpds_close_object(  
  
    kobject object)
```

Input Arguments

`object`
the object to be closed.

Returns

TRUE (1) if object is closed successfully, FALSE (0) otherwise

Description

This function is called on an object when all interaction with the object is complete. In addition to freeing resources that were used to manage the object, this function also writes any component of the data set that has not yet been written and may alter index order and datatype of the data to that is supported by the file format.

If the object was created with the `kpds_reference_object` function call, or if another object was created as a reference of the one being closed, then the object might be sharing some of its resources with other objects. If this is the case, then those shared resources will not be freed, but rather they will be disassociated from the object being closed. Thus, closing an object does not affect any other object.

G.1.8. `kpds_reference_object()` — *create a reference of a data object.*

Synopsis

```
kobject kpds_reference_object(  
    kobject object)
```

Input Arguments

`object`

the object to be referenced.

Returns

a kobject that is a reference of the input object on success, KOBJECT_INVALID upon failure

Description

This function is used to create a reference object for a data object that can be treated as a second independent data object under most circumstances. A referenced object is similar conceptually to a symbolic link in a UNIX file system in most respects. For example, getting data from an input object and a reference of the object will result in the same data. Data that is put on an output object can then be accessed from any of that object's references.

The similarity ends there. Once an object is referenced, the two resulting objects are equivalent--there is no way to distinguish which was the original. In fact, closing the original does not in any way affect the reference, and visa-versa.

kpds_reference_object creates a new object that has presentation attributes that are independent of the original object's presentation attributes. The presentation attributes are UNCOUPLED from the physical attributes, see the description found in Table 4 in Chapter 6 of the the VisiQuest Programmer's Manual on the KPDS_COUPLING attribute for more information. The two objects (or more if there are several calls to kpds_reference_object) share all physical data.

The default index order will be set to KWIDTH, KHEIGHT, KDEPTH, KTIME and KELEMENTS for the value and mask data, to KMAP_WIDTH, KMAP_HEIGHT, KMAP_ELEMENTS, KMAP_DEPTH and KMAP_TIME for the map data and KWIDTH, KHEIGHT, KDEPTH and KDI-MENSION for the location data.

G.1.9. kpds_copy_object() — <i>copy all data and attributes from one object to another.</i>
--

Synopsis

```
int kpds_copy_object(  
  
    kobject source_object,  
    kobject destination_object)
```

Input Arguments

source_object
the object that serves as a source for the data and attributes.

Output Arguments

destination_object

the object that will serve as a destination in the copy operation.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

kpds_copy_object copies all physical & presentation attributes and all data from the source_object to the destination_object. This means that all the attributes and data (not just those that are part of the polymorphic data model) are copied. For example, the source object may contain data and attributes that one of the other services (geometry, numerical, etc.) uses. These will also be copied.

G.1.10. kpds_copy_remaining_data() — *copy remaining data from source to destination*

Synopsis

```
int kpds_copy_remaining_data(  
  
    kobject source_object,  
    kobject destination_object)
```

Input Arguments

source_object
the source object

Output Arguments

destination_object
the destination object

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

kpds_copy_remaining_data copies any data that has not been previously written to in the destination from a specified source.

G.1.11. `kpds_copy_object_attr()` — *copy all presentation attributes from one data object to another.*

Synopsis

```
int
kpds_copy_object_attr(
    kobject source,
    kobject destination)
```

Input Arguments

`source`
the object that serves as the source for the attributes.

Output Arguments

`destination`
the object that serves as the destination for the operation.

Returns

TRUE on success, FALSE otherwise

Description

This function copies all presentation attributes from the `source_object` to the `destination_object`. This means that all the attributes of the object will be copied (not just parts of the polymorphic data model) as well. For example, the source object may contain attributes that one of the other services (geometry, numerical, etc.) uses. These will also be copied. Segments present in the source object will be created in the destination object if they are not already present.

There are three attributes for each data component (i.e. Value, Mask, etc.) that are affected by this function call in a special way: `KPDS*_SIZE`, `KPDS*_DATA_TYPE`, and `KPDS*_INDEX_ORDER`. These attributes are used to define how the data is stored. When this function is called, these attributes will appear to change to the user, but the storage of the data will only be affected if the `KPDS_COUPLING` attribute is set to `KCOUPLED`.

For more information on the behavior of attributes, please refer to `kpds_sync_object`, `kpds_get_attribute` and `kpds_set_attribute`.

G.1.12. `kpds_copy_object_data()` — *copy all data from one object to another object.*

Synopsis

```
int kpds_copy_object_data(  
  
    kobject source_object,  
    kobject destination_object)
```

Input Arguments

`source_object`
the object that serves as a source for the data.

Output Arguments

`destination_object`
the object that will serve as a destination in the copy operation.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function copies all of the data associated with the `source_object` to the `destination_object`. This means that all of the data in the object will be copied, not just the data associated with the polymorphic data model (Value, Mask, Map, Location and Time data). For example, the source object may contain data that one of the other services (such as geometry services or color services.) use. These data will also be copied.

This routine will create all of the data in the `destination_object` that do not exist. It will initialize two attributes that are used to define the physical storage the of the data: `KPDS_*_SIZE`, and `KPDS_*_DATA_TYPE`. If the data already exists in the `destination_object`, then the data will be replaced with what is in the `source_object`. In this case, the data type and size will not be changed. As the data is copied from the source to the destination, it will be cast and resized to fit the destination. The attribute that is used to control how the resize occurs is `KPDS_*_INTERPOLATE`. This attribute can take on the values `KNONE` indicating that no resize should occur (if resize is necessary, then this function will fail); `KPAD`, indicating that the data should be padded with the `KPDS_*_PAD_VALUE` if the source object is larger than the source, or clipped if the destination is smaller than the source; or `KZERO_ORDER`, which indicates that a zero-order hold interpolation (pixel replication) should occur.

G.1.13. <code>kpds_sync_object()</code> — <i>synchronize physical and presentation layers of a data object.</i>
--

Synopsis

```
int kpds_sync_object(  
  
    kobject object,  
    int direction)
```

Input Arguments

`object`

data object to be re-synchronized.

`direction`

the desired direction of the synchronization. the legal values are KPRES2PHYS, which indicates that the physical layer will be updated to correspond to the presentation layer; and KPHYS2PRES, which indicates that the presentation layer will be updated to correspond to the physical layer.

Returns

TRUE (1) if object sync'ed, FALSE (0) otherwise

Description

This function is used to update physical attributes of the entire data object to match those of the presentation layer, or visa-versa.

When an attribute is set via `kpds_set_attribute(s)` or `kpds_copy_object_attr`, the presentation version of the attribute is the only thing that is directly manipulated. The `KPDS_COUPLING` attribute is used at that time to determine if the physical attribute should be updated to correspond to its value at the presentation level or vice versa. The `KPDS_COUPLING` attribute can take on one of three values: `KUNCOUPLED`, `KCOUPLED`, or `KDEMAND`. If it is set to `KUNCOUPLED` or `KDEMAND`, then Polymorphic Services will not update the physical layer. If the attribute is set to `KCOUPLED`, then Polymorphic Services immediately updates the physical layer any time `kpds_set_attribute(s)` is called. If the attribute is set to `KDEMAND`, then this updating will only occur when `kpds_sync_object` is called. If the `KPDS_COUPLING` attribute is set to `KUNCOUPLED`, then this routine will simply return and an error message will be returned.

G.2. Data Functions

- `kpds_get_data()` - retrieve data from a data object.
- `kpds_put_data()` - store data in a data object.

G.2.1. `kpds_get_data()` — *retrieve data from a data object.*

Synopsis

```
kaddr
kpds_get_data(
    kobject object,
    char *primitive,
    kaddr data)
```

Input Arguments

`object`

the object that will serve as the source for the data to be retrieved.

primitive

a description of the unit of data desired.

data

a pointer to the region of memory that will serve as a destination for the data. If this value is NULL, then sufficient space for this operation will be allocated to return the data. The data type `kaddr` is used because it represents a generic data pointer.

Output Arguments

data

if "data" is not initially NULL, then the memory that it points to will be filled with the requested data.

Returns

If "data" is not initially NULL, then the data space pointed to by "data" will be returned on success. If the "data" argument is NULL, then a new pointer to the requested data will be returned. Unsuccessful calls to this routine are indicated by a return value of NULL.

Description

This function is used to retrieve a unit of data, referred to as a "primitive", from a data object.

Data within a data object is accessed in terms of the polymorphic data model. The polymorphic data model contains 5 different data segments: value, mask, location, map, and time. Data is retrieved from an object by using a segment-specific primitive.

The first argument to this function is the data object from which the data should be retrieved.

The second argument is the data primitive which should be retrieved.

Data primitives are accessed relative to the position and offset attributes. For instance, the position and offset at which a value primitive will be accessed are determined by the attributes `KPDS_VALUE_POSITION` and `KPDS_VALUE_OFFSET`.

Successive calls to `kpds_get_data` cause an automatic increment of the position attribute. The position is incremented by the amount of data contained in the primitive. This allows `kpds_get_data` to be called repeatedly in order to traverse all data within a data segment. For example, successive calls to get `KPDS_VALUE_PLANE` primitives will return successive planes down depth, time, and then elements.

Below is a list of the types of primitives that are available:

point - specifies that a single value will be returned. Successive calls to `kpds_get_data` will result in adjacent points being returned, as described above. An example of a point primitive is `KPDS_VALUE_POINT`.

line - specifies that a one-dimensional unit of data will be returned. The direction of a line is always along the width of the dataset. An example of a line primitive is `KPDS_VALUE_LINE`.

- plane** - specifies that a two-dimensional unit of data will be returned. The plane is defined along the width and height of the segment. An example of a plane primitive is KPDS_VALUE_PLANE.
- volume** - specifies that a three-dimensional unit of data will be returned. The volume is defined along the width, height, and depth of the segment. An example of a volume primitive is KPDS_VALUE_VOLUME.
- region** - specifies that a n-dimensional unit of data will be returned. The n varies from segment to segment and is based on the dimensionality of the segment being accessed. For example, the value segment will be 5-dimensional, so n is 5 for the value segment. A region upper corner is specified by the current position (for the value segment, this is a five-tuple). The size of the region is given by a region size attribute (such as KPDS_VALUE_REGION_SIZE) which must be set prior to using this primitive. An example of a region primitive is KPDS_VALUE_REGION.
- all** - specifies that all data for the specified segment should be returned. An example of an all primitive is KPDS_VALUE_ALL. Note that position and offset do not affect this primitive.
- vector** - specifies that a one-dimensional unit of data should be returned. An example of a vector primitive is KPDS_VALUE_VECTOR.

For the five segments each has a different dimension that specifies a vector. Below is a list of the data segments and their associated vector definition:

VALUE - KELEMENTS
 MASK - KELEMENTS
 MAP - KWIDTH
 LOCATION - KDIMENSION
 TIME - KTIME

The third argument, "data", serves as both an input and an output argument. As input, it dictates whether kpds_get_data must allocate space sufficient for the operation. If the argument is NULL, then memory will be allocated to store the data primitive requested. A pointer to that memory will be returned. If this argument is not NULL, then kpds_get_data assumes that the "data" argument is a pointer to a sufficient amount of memory with the correct dimensionality for the primitive (no memory allocation occurs). In this case, if this routine returns successfully, then the return value is the pointer "data".

Restrictions

This routine assumes that if the argument "data" is not NULL, then it contains the appropriate amount of memory with the appropriate dimensionality for the requested primitive.

G.2.2. `kpds_put_data()` — *store data in a data object.*

Synopsis

```
int
kpds_put_data(
    kobject object,
    char *primitive,
    kaddr data)
```

Input Arguments

`object`
the data object that will serve as a destination for the data.

`primitive`
a description of the unit of data in the argument "data".

`data`
a pointer to the region of memory that will be stored.

Returns

TRUE on success, FALSE otherwise

Description

This function is used to store a unit of data, referred to as a "primitive", into a data object.

Data within a data object is accessed in terms of the polymorphic data model. The polymorphic data model contains 5 different data segments: value, mask, location, map, and time. Data is stored in an object by using a segment-specific primitive.

The first argument to this function is the data object in which the data should be stored.

The second argument is the data primitive which should be stored.

Data primitives are accessed relative to the position and offset attributes. For instance, the position and offset at which a value primitive will be accessed are determined by the attributes `KPDS_VALUE_POSITION` and `KPDS_VALUE_OFFSET`.

Successive calls to `kpds_get_data` cause an automatic increment of the position attribute. The position is incremented by the amount of data contained in the primitive. This allows `kpds_put_data` to be called repeatedly in order to traverse all data within a data segment. For example, successive calls to get `KPDS_VALUE_PLANE` primitives will store successive planes down depth, time, and then

elements.

Below is a list of the types of primitives that are available:

- point** - specifies that a single value will be stored. Successive calls to `kpds_put_data` will result in adjacent points being stored, as described above. An example of a point primitive is `KPDS_VALUE_POINT`.
- line** - specifies that a one-dimensional unit of data will be stored. The direction of a line is always along the width of the dataset. An example of a line primitive is `KPDS_VALUE_LINE`.
- plane** - specifies that a two-dimensional unit of data will be stored. The plane is defined along the width and height of the segment. An example of a plane primitive is `KPDS_VALUE_PLANE`.
- volume** - specifies that a three-dimensional unit of data will be stored. The volume is defined along the width, height, and depth of the segment. An example of a volume primitive is `KPDS_VALUE_VOLUME`.
- region** - specifies that a n-dimensional unit of data will be stored. The n varies from segment to segment and is based on the dimensionality of the segment being accessed. For example, the value segment will be 5-dimensional, so n is 5 for the value segment. A region upper corner is specified by the current position (for the value segment, this is a five-tuple). The size of the region is given by a region size attribute (such as `KPDS_VALUE_REGION_SIZE`) which must be set prior to using this primitive. An example of a region primitive is `KPDS_VALUE_REGION`.
- all** - specifies that all data for the specified segment should be stored. An example of an all primitive is `KPDS_VALUE_ALL`. Note that position and offset do not affect this primitive.
- vector** - specifies that a one-dimensional unit of data will be stored. An example of a vector primitive is `KPDS_VALUE_VECTOR`.

For the five segments each has a different dimension that specifies a vector. Below is a list of the data segments and their associated vector definition:

VALUE - KELEMENTS
MASK - KELEMENTS
MAP - KWIDTH
LOCATION - KDIMENSION
TIME - KTIME

The third argument is the data to be stored. This must be a non-NULL pointer to valid data of the appropriate size (as defined, for example, by the KPDS_VALUE_SIZE attribute) and data type (as defined, for example, by the KPDS_VALUE_DATA_TYPE attribute).

The data is copied out of the data array on storage, so it can be overwritten or freed after the `kpds_put_data` call.

G.3. Attribute Functions

- `kpds_copy_attribute()` - copy an attribute from one object to another
- `kpds_copy_attributes()` - copy multiple attributes from one object to another.
- `kpds_get_attribute()` - get the value of an attribute from a data object
- `kpds_get_attributes()` - get the values of multiple attributes from a data object
- `kpds_match_attribute()` - returns TRUE if the same attribute in two objects match.
- `kpds_match_attributes()` - returns true if the list of segment attributes in two objects match.
- `kpds_print_attribute()` - print the value of an attribute from a data object.
- `kpds_query_attribute()` - get information about an attribute
- `kpds_set_attribute()` - set the values of an attribute in a data object
- `kpds_set_attributes()` - set the values of multiple attributes in a data object.

G.3.1. `kpds_copy_attribute()` — *copy an attribute from one object to another*

Synopsis

```
int
kpds_copy_attribute(
    kobject object1,
    kobject object2,
    char *attribute)
```

Input Arguments

```
object1
    the object to copy from
object2
    the object to copy to
attribute
    the attribute to copy
```

Returns

TRUE on success, FALSE otherwise

Description

This function is used to copy a single attribute from one object to another object.

Data Services manages two versions of some of the attributes associated with each object. These attributes are the size and data type. Internally, the two versions of these attributes are referred to as the physical attribute and the presentation attribute. Typically, the programmer has access to only the presentation versions of these attributes. The physical attributes are set indirectly depending on the setting of KPDS_COUPLING. See `kpds_get_data` for a description of how the presentation and physical attributes affect interaction with the data object.

Other attributes are classified as either shared or unshared. Shared attributes are stored at the physical layer of the attribute, and thus can be shared by multiple references of the data object (see `kpds_reference_object` for more information about references). Unshared attributes, on the other hand, can only be used by the local object (see `kpds_query_attribute` for information on how to determine whether an attribute is shared or unshared).

This function accepts a source object, a destination object, and an attribute name. If the attribute exists in the source object, then it will be copied to the destination object. If the attribute does not exist in the source object, then an error condition is returned.

A complete list of the polymorphic attributes can be found in Chapter 2 of Programming Services Volume II.

G.3.2. `kpds_copy_attributes()` — *copy multiple attributes from one object to another.*

Synopsis

```
int
kpds_copy_attributes(
    kobject object1,
    kobject object2,
    kvalist)
```

Input Arguments

```
object1
    the object to copy from
object2
    the object to copy to va_alist - NULL terminated list of attribute names to copy.
```

Returns

TRUE on success, FALSE otherwise

Description

This function is used to copy multiple attributes from one object to another. The attributes should be provided in a NULL terminated variable argument list.

Data Services manages two versions of some of the attributes associated with each object. These

attributes are the size and data type. Internally, the two versions of these attributes are referred to as the physical attribute and the presentation attribute. Typically, the programmer has access to only the presentation versions of these attributes. The physical attributes are set indirectly depending on the setting of KPDS_COUPLING. See `kpds_get_data` for a description of how the presentation and physical attributes affect interaction with the data object.

Other attributes are classified as either shared or unshared. Shared attributes are stored at the physical layer of the attribute, and thus can be shared by multiple references of the data object (see `kpds_reference_object` for more information about references). Unshared attributes, on the other hand, can only be used by the local object (see `kpds_query_attribute` for information on how to determine whether an attribute is shared or unshared).

This function accepts a source object, a destination object, and a list of attribute names. The last argument to this function must be `NULL`. If it is not `NULL`, then the behavior of this function will be unpredictable (the `NULL` argument is used as a sentinel to indicate the end of the variable argument list. If the sentinel is not present, then this function will continue to attempt to pull arguments off of the stack, until it finds a `NULL`). If each of the attributes exist in the source object, then it will be copied to the destination object. If an attribute does not exist in the source object, then an error condition is returned. In the event that an attribute does not exist, then the remainder of the attributes on the list will not be copied.

A complete list of the polymorphic attributes can be found in Chapter 2 of Programming Services Volume II.

G.3.3. `kpds_get_attribute()` — *get the value of an attribute from a data object*

Synopsis

```
int
kpds_get_attribute(
    kobject object,
    char *attribute,
    kvalist)
```

Input Arguments

`object`

the object from which to retrieve the specified attribute. This must be a legal `kobject` that has been opened or instantiated by an appropriate `kpds` function call, such as `kpds_open_object` or `kpds_reference`.

`attribute`

the name of the attribute to retrieve. this is a character string that is the name of an existing attribute in the object. There are a large number of predefined KPDS attributes. Users can also create attributes via the `kpds_create_attribute` function call.

Returns

TRUE if the attribute was successfully retrieved, FALSE otherwise.

Description

This routine is used to get the value of an attribute from a data object.

Data Services manages two versions of some of the attributes associated with each object. These attributes are the size and data type. Internally, the two versions of these attributes are referred to as the physical attribute and the presentation attribute. Typically, the programmer has access to only the presentation versions of these attributes. The physical attributes are set indirectly depending on the setting of KPDS_COUPLING. See `kpds_get_data` for a description of how the presentation and physical attributes affect interaction with the data object.

Other attributes are classified as either shared or unshared. Shared attributes are stored at the physical layer of the attribute, and thus can be shared by multiple references of the data object (see `kpds_reference_object` for more information about references). Unshared attributes, on the other hand, can only be used by the local object (see `kpds_query_attribute` for information on how to determine whether an attribute is shared or unshared).

This function accepts an object, the name of the attribute that is desired and the address of a location in which to return the value of the attribute. Certain KPDS attributes require more than one argument in order to return the entire attribute. For example, `KPDS_VALUE_SIZE` requires five (5) arguments. Getting the `KPDS_VALUE_DATA_TYPE` attribute might look like this:

```
kpds_get_attribute(object, KPDS_VALUE_DATA_TYPE, &typ);
```

Getting the `KPDS_VALUE_SIZE` attribute, is a little more complex:

```
kpds_get_attribute(object, KPDS_VALUE_SIZE,  
                  &w, &h, &d, &t, &e);
```

A complete list of the polymorphic attributes can be found in Chapter 2 of Programming Services Volume II.

G.3.4. `kpds_get_attributes()` — *get the values of multiple attributes from a data object*

Synopsis

```
int  
kpds_get_attributes(  
    kobject object,  
    kvalist)
```

Input Arguments

`object`

the object from which to retrieve the specified attribute. This must be a legal kobject that has been opened or instantiated by an appropriate kpds function call, such as `kpds_open_object` or `kpds_reference`.

Returns

TRUE if the attribute was successfully retrieved, FALSE otherwise.

Description

This function is used to retrieve the values of an arbitrary number of attributes from a data object.

Data Services manages two versions of some of the attributes associated with each object. These attributes are the size and data type. Internally, the two versions of these attributes are referred to as the physical attribute and the presentation attribute. Typically, the programmer has access to only the presentation versions of these attributes. The physical attributes are set indirectly depending on the setting of `KPDS_COUPLING`. See `kpds_get_data` for a description of how the presentation and physical attributes affect interaction with the data object.

Other attributes are classified as either shared or unshared. Shared attributes are stored at the physical layer of the attribute, and thus can be shared by multiple references of the data object (see `kpds_reference_object` for more information about references). Unshared attributes, on the other hand, can only be used by the local object (see `kpds_query_attribute` for information on how to determine whether an attribute is shared or unshared).

This function accepts an object followed by a list of arguments that alternate between an attribute name and an address into which the attribute's value will be stored. The last argument on this list is a NULL which serves as a flag that indicates that no more attributes are present on the list. Certain KPDS attributes require more than one address in order to return the entire attribute. For example, `KPDS_VALUE_SIZE` requires five (5) arguments. For example,

```
kpds_get_attributes(object,  
    KPDS_VALUE_DATA_TYPE, &typ,  
    KPDS_VALUE_SIZE, &w, &h, &d, &t, &e,  
    NULL);
```

The last argument to this function must be NULL. If it is not NULL, then the behavior of this function will be unpredictable (the NULL argument is used as a sentinel to indicate the end of the variable argument list. If the sentinel is not present, then this function will continue to attempt to pull arguments off of the stack, until it finds a NULL).

A complete list of the polymorphic attributes can be found in Chapter 2 of Programming Services Volume II.

G.3.5. `kpds_match_attribute()` — *returns TRUE if the same attribute in two objects match.*

Synopsis

```
int
kpds_match_attribute(
    kobject object1,
    kobject object2,
    char *attr)
```

Input Arguments

`object1`
the first object on which to match the specified attribute

`object2`
the second object on which to match the specified attribute - the attribute that will be compared in the two objects.

Returns

There are three ways for this routine to return a FALSE: (1) if the attribute in the two objects does not match; (2) if either object does not contain the specified attribute; (3) an error condition resulting from an invalid object or segment. If none of these three conditions exist, then this function will return TRUE.

Description

The purpose of this routine is to allow the programmer to compare a single in two data objects.

Data Services manages two versions of some of the attributes associated with each object. These attributes are the size and data type. Internally, the two versions of these attributes are referred to as the physical attribute and the presentation attribute. Typically, the programmer has access to only the presentation versions of these attributes. The physical attributes are set indirectly depending on the setting of `KPDS_COUPLING`. See `kpds_get_data` for a description of how the presentation and physical attributes affect interaction with the data object.

Other attributes are classified as either shared or unshared. Shared attributes are stored at the physical layer of the attribute, and thus can be shared by multiple references of the data object (see `kpds_reference_object` for more information about references). Unshared attributes, on the other hand, can only be used by the local object (see `kpds_query_attribute` for information on how to determine whether an attribute is shared or unshared).

This routine will return TRUE if the specified attribute has the same value in both of the objects. This routine will return FALSE if the attribute does not have the same value in both of the objects. `kpds_match_attribute` will also return FALSE if the attribute does not exist in either or both of the objects.

A complete list of the polymorphic attributes can be found in Chapter 2 of Programming Services Volume II.

G.3.6. `kpds_match_attributes()` — *returns true if the list of segment attributes in two objects match.*

Synopsis

```
int
kpds_match_attributes(
    kobject object1,
    kobject object2,
    kvalist)
```

Input Arguments

`object1`
the first object on which to match the specified attributes

`object2`
the second object on which to match the specified attributes `va_alist` - variable argument list, that contains an arbitrarily long list of attributes followed a NULL. It takes the form:

```
ATTRIBUTE_NAME1, ATTRIBUTE_NAME2, ..., NULL
```

Returns

There are three ways for this routine to return a FALSE: (1) if any of the attributes between the two objects do not match; (2) if either object does not contain one or more of the specified attributes; (3) an error condition resulting from an invalid object or segment. If none of these three conditions exist, then this function will return TRUE.

Description

The purpose of this routine is to allow the programmer to compare multiple attributes in two object.

Data Services manages two versions of some of the attributes associated with each object. These attributes are the size and data type. Internally, the two versions of these attributes are referred to as the physical attribute and the presentation attribute. Typically, the programmer has access to only the presentation versions of these attributes. The physical attributes are set indirectly depending on the setting of `KPDS_COUPLING`. See `kpds_get_data` for a description of how the presentation and physical attributes affect interaction with the data object.

Other attributes are classified as either shared or unshared. Shared attributes are stored at the physical layer of the attribute, and thus can be shared by multiple references of the data object (see `kpds_reference_object` for more information about references). Unshared attributes, on the other hand, can only be used by the local object (see `kpds_query_attribute` for information on how to determine whether an attribute is shared or unshared).

This routine will return TRUE if all of the specified attributes have the same value in the objects. This routine will return FALSE if any of the attributes do not match `kpds_match_attributes` will also return FALSE if any of the attributes do not exist in either or both of the two objects.

The last argument to this function must be NULL. If it is not NULL, then the behavior of this function will be unpredictable (the NULL argument is used as a sentinel to indicate the end of the variable argument list. If the sentinel is not present, then this function will continue to attempt to pull arguments off of the stack, until it finds a NULL).

A complete list of the polymorphic attributes can be found in Chapter 2 of Programming Services Volume II.

G.3.7. `kpds_print_attribute()` — *print the value of an attribute from a data object.*

Synopsis

```
int
kpds_print_attribute(
    kobject object,
    char *attribute,
    kfile *printfile)
```

Input Arguments

`object`
the object containing the attribute

`attribute`
the attribute to print

`printfile`
the open kfile to print to

Returns

TRUE on success, FALSE otherwise

Description

This function is used to print the value of an attribute from a data object to an output file.

This function is typically used by such programs as `kprdata` to print out the values of attributes in an object.

G.3.8. `kpds_query_attribute()` — *get information about an attribute*

Synopsis

```
int
kpds_query_attribute(
```

```
kobject object,  
char *attribute,  
int *num_args,  
int *arg_size,  
int *data_type,  
int *permanent)
```

Input Arguments

object
the object with the attribute

attribute
name of the attribute to be queried.

num_args
number of arguments in this attribute

arg_size
size of each argument in this attribute.

data_type
datatype of the attribute

permanent
is the attribute stored or transient? The return value will be either TRUE or FALSE

Returns

TRUE if attribute exists, FALSE otherwise

Description

This function is used for two purposes: (1) to determine the existence of an attribute; and (2) to obtain the characteristics of the attribute.

Data Services manages two versions of some of the attributes associated with each object. These attributes are the size and data type. Internally, the two versions of these attributes are referred to as the physical attribute and the presentation attribute. Typically, the programmer has access to only the presentation versions of these attributes. The physical attributes are set indirectly depending on the setting of KPDS_COUPLING. See `kpds_get_data` for a description of how the presentation and physical attributes affect interaction with the data object.

Other attributes are classified as either shared or unshared. Shared attributes are stored at the physical layer of the attribute, and thus can be shared by multiple references of the data object (see `kpds_reference_object` for more information about references). Unshared attributes, on the other hand, can only be used by the local object.

The difference between shared and unshared attributes is abstracted from the user at the PDS level. The permanent attributes are generally shared, and the non-permanent attributes are generally non-shared. Permanent attributes are attributes that will be stored as part of an output object when it is written. Any attributes that are retrieved when an object is opened are also permanent attributes. Non-permanent attributes exist only while the program that is operating on the object is executing.

The datatype argument indicates what kind of information is stored in the attribute. Attributes can be one of the following data types: KBYTE, KUBYTE, KSHORT, KUSHORT, KINT, KUINT, KLONG,

KULONG, KFLOAT, KDOUBLE, KCOMPLEX, or KDCOMPLEX.

The num_args argument indicates how many arguments must be passed in an argument list to one of the attribute functions.

The size arguments indicates the number of units of the data type there are in each argument. This argument allows arrays of information to be stored as attributes.

Any arguments after object, segment, and attribute name can be set to NULL if the user does not need these values.

G.3.9. kpds_set_attribute() — *set the values of an attribute in a data object*

Synopsis

```
int
kpds_set_attribute(
    kobject object,
    char *attribute,
    kvalist)
```

Input Arguments

object

the object in which to set the specified attribute. This must be a legal kobject that has been opened or instantiated by an appropriate kpds function call, such as kpds_open_object or kpds_reference.

attribute

the name of the attribute to set. this is a character string that is the name of an existing attribute in the object. There are a large number of predefined KPDS attributes. Users can also create attributes via the kpds_create_attribute function call. va_alist - a C variable argument list that contains value or values of the specified attribute.

Returns

TRUE if the attribute was successfully set, FALSE otherwise.

Description

This function is used to assign the value of a attribute to a data object.

Data Services manages two versions of some of the attributes associated with each object. These attributes are the size and data type. Internally, the two versions of these attributes are referred to as the physical attribute and the presentation attribute. Typically, the programmer has access to only the presentation versions of these attributes. The physical attributes are set indirectly depending on the setting of KPDS_COUPLING. See kpds_get_data for a description of how the presentation and physical attributes affect interaction with the data object.

Other attributes are classified as either shared or unshared. Shared attributes are stored at the physical layer of the attribute, and thus can be shared by multiple references of the data object (see `kpds_reference_object` for more information about references). Unshared attributes, on the other hand, can only be used by the local object (see `kpds_query_attribute` for information on how to determine whether an attribute is shared or unshared).

This function accepts an object, the name of the attribute that is desired and the value of the attribute. Certain KPDS attributes require more than one value in order to set the entire attribute. For example, `KPDS_VALUE_SIZE` requires five (5) arguments. For example, setting the `KPDS_VALUE_DATA_TYPE` attribute might look like this:

```
kpds_set_attribute(object, KPDS_VALUE_DATA_TYPE, KFLOAT);
```

Setting the `KPDS_VALUE_SIZE` attribute, is a little more complex:

```
kpds_set_attribute(object, KPDS_VALUE_SIZE,  
                  100, 100, 1, 1, 1);
```

A complete list of the polymorphic attributes can be found in Chapter 2 of Programming Services Volume II.

G.3.10. <code>kpds_set_attributes()</code> — <i>set the values of multiple attributes in a data object.</i>
--

Synopsis

```
int  
kpds_set_attributes(  
    kobject object,  
    kvalist)
```

Input Arguments

`object`
the object in which to set the specified attribute. This must be a legal `kobject` that has been opened or instantiated by an appropriate `kpds` function call, such as `kpds_open_object` or `kpds_reference`.

Returns

TRUE if the attribute was successfully retrieved, FALSE otherwise.

Description

This function is used to assign the values of an arbitrary number of attributes to a data object.

Data Services manages two versions of some of the attributes associated with each object. These attributes are the size and data type. Internally, the two versions of these attributes are referred to as the physical attribute and the presentation attribute. Typically, the programmer has access to only the presentation versions of these attributes. The physical attributes are set indirectly depending on the setting of `KPDS_COUPLING`. See `kpds_get_data` for a description of how the presentation and physical attributes affect interaction with the data object.

Other attributes are classified as either shared or unshared. Shared attributes are stored at the physical layer of the attribute, and thus can be shared by multiple references of the data object (see `kpds_reference_object` for more information about references). Unshared attributes, on the other hand, can only be used by the local object (see `kpds_query_attribute` for information on how to determine whether an attribute is shared or unshared).

This function accepts an object followed by a list of arguments that alternate between an attribute name and values that the specified attribute is to be set to. The last argument on this list is a `NULL` which serves as a flag that indicates that no more attributes are present on the list. Certain KPDS attributes require more than one value in order to return the entire attribute. For example, `KPDS_VALUE_SIZE` requires five (5) arguments. For example,

```
kpds_set_attributes(object,  
    KPDS_VALUE_DATA_TYPE, KFLOAT,  
    KPDS_VALUE_SIZE, 100, 100, 1, 1, 1,  
    NULL);
```

The last argument to this function must be `NULL`. If it is not `NULL`, then the behavior of this function will be unpredictable (the `NULL` argument is used as a sentinel to indicate the end of the variable argument list. If the sentinel is not present, then this function will continue to attempt to pull arguments off of the stack, until it finds a `NULL`).

A complete list of the polymorphic attributes can be found in Chapter 2 of Programming Services Volume II.

G.4. Location Functions

- `kpds_copy_location()` - copy the location segment from one object to another.
- `kpds_copy_location_attr()` - copy all location attributes from one object to another object.
- `kpds_copy_location_data()` - copy all location data from one object to another object.
- `kpds_create_location()` - create a location segment within a data object.
- `kpds_destroy_location()` - destroy the location segment in a data object.
- `kpds_query_location()` - determine if the location segment exists in a data object.

G.4.1. `kpds_copy_location()` — *copy the location segment from one object to another.*

Synopsis

```
int kpds_copy_location(  
    kobject object1,  
    kobject object2,  
    int      copy_through_presentation)
```

Input Arguments

`object1`
the object that serves as a source for the location segment.

`copy_through_presentation`
if set to TRUE, the copy will be performed through the presentation of the source and destination objects. if set to FALSE, the copy will be a direct copy of the physical data.

Output Arguments

`object2`
the object that will serve as a destination for the copy operation.

Returns

TRUE if copy was successful, FALSE otherwise

Description

This function copies all of the attributes and data contained in the location segment of one object into the location segment contained in another object. If the location segment exists in the destination object, then its data will be replaced with the data from the source object. If the location segment does not exist in the destination object, it will be created.

All location segment attributes will be copied from one object to the other. This includes presentation attributes, such as position and offset.

The location data can be optionally copied through the presentations of the two data objects. This implies that any presentation stages which are normally invoked during a `kpds_get_data` on the source object or a `kpds_put_data` call on the destination object will be used for the copy.

Any of the following presentation stages can be invoked on either the source object or the destination object : casting, scaling, normalizing, padding, and interpolating.

The following presentation stages can be invoked on only the source object : axis assignment, position, and offset. The resulting copy will be transposed to be in the default axis ordering. The resulting copy will also appear to be shifted by the source position and offset.

The resulting copy will contain data in the most explicit grid type between the source and the

destination.

For example, if the destination grid type is curvilinear, and the source grid type is uniform, the resulting copy will contain curvilinear data. However, if the destination grid type is uniform, and the source grid type is curvilinear, then the destination must be curvilinear. In general, the copy can not contain less data than is contained in the source.

If the destination grid type is not set, then the copy will preserve the source grid type.

These presentation stages are brought into the data pipeline by setting the appropriate presentation attributes on the data objects. Please see the chapter on polymorphic data services in the data services manual for more information on the various presentation attributes.

The following example may help in visualizing the process. The source object has a presentation data type different from the physical data type, and a presentation size different from the physical size. The destination object only has a different presentation and physical data type.

```
source -> interpolating -> casting -|...| -> casting -> destination
```

In the above example, the data resulting from the copy will be interpolated to a new size and cast to a new data type. When being copied into the destination object, the data will be cast yet again before it finally is stored.

If the presentation and physical layers of the destination object are coupled then the destination attributes will be ignored and the source presentation attributes will be propagated to the destination physical layer. The copy, in essence, will be performed only through the source presentation, with the destination physical and presentation layers taking on the characteristics of the source presentation. By default, output objects are not coupled, so this behavior is typical.

The copy need not be performed using the presentation. If the data is not copied through the presentation, the destination data will be a direct copy of the physical source data. The physical size and data type of the location segment will be reflected in the destination data object from the source to the destination since they describe the physical state of the data. The grid type of the destination will always match the grid type of the source in this case.

This function is equivalent to performing successive calls to `kpds_copy_location_attr` and `kpds_copy_location_data`.

G.4.2. `kpds_copy_location_attr()` — *copy all location attributes from one object to another object.*

Synopsis

```
int kpds_copy_location_attr(  
    kobject object1,
```

```
kobject object2)
```

Returns

TRUE if copy was successful, FALSE otherwise

Description

This function copies all the location attributes from one object to another object.

If the destination object does not contain a location segment, then this function will create a location segment, and initialize its size and data type attributes to those in the source object. If the location data already exists in the destination object, then the presentation attributes will be set to the source object's settings.

The destination's physical attributes will change depending on the coupling of the data objects. If the destination is coupled, then the physical attributes will be changed as well. Any data contained in the destination will be cast, rescaled, or resized to match its new physical attributes.

G.4.3. `kpds_copy_location_data()` — *copy all location data from one object to another object.*

Synopsis

```
int kpds_copy_location_data(  
    kobject object1,  
    kobject object2,  
    int      copy_through_presentation)
```

Input Arguments

`object1`
the object that serves as a source for the data.

`copy_through_presentation`
if set to TRUE, the copy will be performed through the presentation of the source and destination objects. if set to FALSE, the copy will be a direct copy of the physical data.

Output Arguments

`object2`
the object that will serve as a destination for the copy operation.

Returns

TRUE if copy was successful, FALSE otherwise

Description

This function copies all of the data contained in the location segment of one object into the location segment contained in another object. If the location segment exists in the destination object, then its data will be replaced with the data from the source object. If the location segment does not exist in the destination object, it will be created.

The location data can be optionally copied through the presentations of the two data objects. This implies that any presentation stages which are normally invoked during a `kpds_get_data` on the source object or a `kpds_put_data` call on the destination object will be used for the copy.

Any of the following presentation stages can be invoked on either the source object or the destination object : casting, scaling, normalizing, padding, and interpolating.

The following presentation stages can be invoked on only the source object : axis assignment, position, and offset. The resulting copy will be transposed to be in the default axis ordering. The resulting copy will also appear to be shifted by the source position and offset.

The resulting copy will contain data in the most explicit grid type between the source and the destination.

For example, if the destination grid type is curvilinear, and the source grid type is uniform, the resulting copy will contain curvilinear data. However, if the destination grid type is uniform, and the source grid type is curvilinear, then the destination must be curvilinear. In general, the copy can not contain less data than is contained in the source.

If the destination grid type is not set, then the copy will preserve the source grid type.

These presentation stages are brought into the data pipeline by setting the appropriate presentation attributes on the data objects. Please see the chapter on polymorphic data services in the data services manual for more information on the various presentation attributes.

The following example may help in visualizing the process. The source object has a presentation data type different from the physical data type, and a presentation size different from the physical size. The destination object only has a different presentation and physical data type.

source -> interpolating -> casting -[...]-> casting -> destination

In the above example, the data resulting from the copy will be interpolated to a new size and cast to a new data type. When being copied into the destination object, the data will be cast yet again before it finally is stored.

If the presentation and physical layers of the destination object are coupled then the destination attributes will be ignored and the source presentation attributes will be propagated to the destination physical layer. The copy, in essence, will be performed only through the source presentation, with the destination physical and presentation layers taking on the characteristics of the source presentation. By default, output objects are not coupled, so this behavior is typical.

The copy need not be performed using the presentation. If the data is not copied through the

presentation, the destination data will be a direct copy of the physical source data. The physical size and data type of the location segment will be reflected in the destination data object from the source to the destination since they describe the physical state of the data. The grid type of the destination will always match the grid type of the source in this case.

G.4.4. `kpds_create_location()` — *create a location segment within a data object.*

Synopsis

```
int kpds_create_location(  
    kobject object)
```

Input Arguments

`object`
object in which to create the location segment.

Returns

TRUE if the location segment was successfully created, FALSE otherwise

Description

This function is used to create a location segment within a specified data object. The size of the location segment will be initialized to match any the sizes shared with any other polymorphic segments.

Either uniform, rectilinear, or curvilinear location grids can be created. The attribute `KPDS_LOCATION_GRID` should be set to the desired grid type of either `KUNIFORM`, `KRECTILINEAR`, or `KCURVILINEAR`, before calling `kpds_create_location()`. By default, if the grid attribute is not set, curvilinear location will be created.

It is considered an error to create a location segment if the object already contains one.

G.4.5. `kpds_destroy_location()` — *destroy the location segment in a data object.*

Synopsis

```
int kpds_destroy_location(  
    kobject object)
```

Input Arguments

`object`
object from which to remove the location segment.

Returns

TRUE if the location segment is successfully destroyed, FALSE otherwise

Description

This function is used to destroy the location segment contained within an object. Once the location segment has been destroyed, any data or attributes associated with the location data will be lost forever. A new location segment can be created in its place with the function `kpds_create_location`.

If the location segment does not exist in the specified object, it is considered to be an error.

G.4.6. `kpds_query_location()` — *determine if the location segment exists in a data object.*

Synopsis

```
int kpds_query_location(  
    kobject object)
```

Input Arguments

`object`
data object to be queried.

Returns

TRUE if the location segment exists, FALSE otherwise

Description

This function is used to determine if the location segment exists in a data object. If location segment exists in the specified object, then this function will return TRUE. If the object is invalid, or location data does not exist in the object, then this function will return FALSE.

G.5. Map Functions

- `kpds_copy_map()` - copy the map segment from one object to another.
- `kpds_copy_map_attr()` - copy all map attributes from one object to another object.
- `kpds_copy_map_data()` - copy all map data from one object to another object.
- `kpds_create_map()` - create a map segment within a data object.
- `kpds_destroy_map()` - destroy the map segment in a data object.
- `kpds_query_map()` - determine if the map segment exists in a data object.

G.5.1. `kpds_copy_map()` — *copy the map segment from one object to another.*

Synopsis

```
int kpds_copy_map(  
    kobject object1,  
    kobject object2,  
    int      copy_through_presentation)
```

Input Arguments

`object1`
the object that serves as a source for the map segment.

`copy_through_presentation`
if set to TRUE, the copy will be performed through the presentation of the source and destination objects. if set to FALSE, the copy will be a direct copy of the physical data.

Output Arguments

`object2`
the object that will serve as a destination for the copy operation.

Returns

TRUE if copy was successful, FALSE otherwise

Description

This function copies all of the attributes and data contained in the map segment of one object into the map segment contained in another object. If the map segment exists in the destination object, then its data will be replaced with the data from the source object. If the map segment does not exist in the destination object, it will be created.

All map segment attributes will be copied from one object to the other. This includes presentation attributes, such as position and offset.

If the mapping mode is set to map the data, the map segment will appear not to be present and this call will fail.

The map data can be optionally copied through the presentations of the two data objects. This implies that any presentation stages which are normally invoked during a `kpds_get_data` on the source object or a `kpds_put_data` call on the destination object will be used for the copy.

Any of the following presentation stages can be invoked on either the source object or the destination object : casting, scaling, normalizing, padding, and interpolating.

The following presentation stages can be invoked on only the source object : axis assignment, position, and offset. The resulting copy will be transposed to be in the default axis ordering. The resulting copy

will also appear to be shifted by the source position and offset.

These presentation stages are brought into the data pipeline by setting the appropriate presentation attributes on the data objects. Please see the chapter on polymorphic data services in the data services manual for more information on the various presentation attributes.

The following example may help in visualizing the process. The source object has a presentation data type different from the physical data type and a presentation size different from the physical size. The destination object only has a different presentation and physical data type.

```
source -> interpolating -> casting -[...] -> casting -> destination
```

In the above example, the data resulting from the copy will be interpolated to a new size and cast to a new data type before being copied into the destination object. When being copied into the destination object, the data will be cast yet again before it finally is stored.

If the presentation and physical layers of the destination object are coupled then the destination attributes will be ignored and the source presentation attributes will be propagated to the destination physical layer. The copy, in essence, will be performed only through the source presentation, with the destination physical and presentation layers taking on the characteristics of the source presentation. By default, output objects are not coupled, so this behavior is typical.

The copy need not be performed using the presentation. If the data is not copied through the presentation, the destination data will be a direct copy of the physical source data. The physical size and data type of the map segment will be reflected in the destination data object from the source to the destination since they describe the physical state of the data.

This function is equivalent to performing successive calls to `kpds_copy_map_attr` and `kpds_copy_map_data`.

G.5.2. <code>kpds_copy_map_attr()</code> — <i>copy all map attributes from one object to another object.</i>

Synopsis

```
int kpds_copy_map_attr(  
    kobject object1,  
    kobject object2)
```

Returns

TRUE if copy was successful, FALSE otherwise

Description

This function copies all the map attributes from one object to another object.

If the destination object does not contain a map segment, then this function will create a map segment, and initialize its size and data type attributes to those in the source object. If the map data already exists in the destination object, then the presentation attributes will be set to the source object's settings.

The destination's physical attributes will change depending on the coupling of the data objects. If the destination is coupled, then the physical attributes will be changed as well. Any data contained in the destination will be cast, rescaled, or resized to match its new physical attributes.

If the mapping mode is set to map the data. then this function will not copy the map attributes.

G.5.3. `kpds_copy_map_data()` — *copy all map data from one object to another object.*

Synopsis

```
int kpds_copy_map_data(  
    kobject object1,  
    kobject object2,  
    int      copy_through_presentation)
```

Input Arguments

`object1`

the object that serves as a source for the data.

`copy_through_presentation`

if set to TRUE, the copy will be performed through the presentation of the source and destination objects. if set to FALSE, the copy will be a direct copy of the physical data.

Output Arguments

`object2`

the object that will serve as a destination for the copy operation.

Returns

TRUE if copy was successful, FALSE otherwise

Description

This function copies all of the data contained in the map segment of one object into the map segment contained in another object. If the map segment exists in the destination object, then its data will be replaced with the data from the source object. If the map segment does not exist in the destination object, it will be created.

The map data can be optionally copied through the presentations of the two data objects. This implies that any presentation stages which are normally invoked during a `kpds_get_data` on the source object or a `kpds_put_data` call on the destination object will be used for the copy.

When copying through the presentation, if the mapping mode is set to map the data, the map segment will appear not to be present and this call will fail. The copy will work normally if not copying through the presentations.

Any of the following presentation stages can be invoked on either the source object or the destination object : casting, scaling, normalizing, padding, and interpolating.

The following presentation stages can be invoked on only the source object : axis assignment, position, and offset. The resulting copy will be transposed to be in the default axis ordering. The resulting copy will also appear to be shifted by the source position and offset.

These presentation stages are brought into the data pipeline by setting the appropriate presentation attributes on the data objects. Please see the chapter on polymorphic data services in the data services manual for more information on the various presentation attributes.

The following example may help in visualizing the process. The source object has a presentation data type different from the physical data type and a presentation size different from the physical size. The destination object only has a different presentation and physical data type.

```
source -> interpolating -> casting -[...]-> casting -> destination
```

In the above example, the data resulting from the copy will be interpolated to a new size and cast to a new data type before being copied into the destination object. When being copied into the destination object, the data will be cast yet again before it finally is stored.

If the presentation and physical layers of the destination object are coupled then the destination attributes will be ignored and the source presentation attributes will be propagated to the destination physical layer. The copy, in essence, will be performed only through the source presentation, with the destination physical and presentation layers taking on the characteristics of the source presentation. By default, output objects are not coupled, so this behavior is typical.

The copy need not be performed using the presentation. If the data is not copied through the presentation, the destination data will be a direct copy of the physical source data. The physical size and data type of the map segment will be reflected in the destination data object from the source to the destination since they describe the physical state of the data. Again, if not copying through the presentation, the copy will ignore the mapping mode of the object.

G.5.4. `kpds_create_map()` — *create a map segment within a data object.*

Synopsis

```
int kpds_create_map(  
    kobject object)
```

Input Arguments

`object`
object in which to create the map segment.

Returns

TRUE if the map segment was successfully created, FALSE otherwise

Description

This function is used to create a map segment within a specified data object. The map segment can not

be created if mapping mode is set to map the value data.

It is considered an error to create a map segment if the object already contains one.

G.5.5. `kpds_destroy_map()` — *destroy the map segment in a data object.*

Synopsis

```
int kpds_destroy_map(  
    kobject object)
```

Input Arguments

`object`
object from which to remove the map segment.

Returns

TRUE if the map segment is successfully destroyed, FALSE otherwise

Description

This function is used to destroy the map segment contained within an object. Once the map segment has been destroyed, any data or attributes associated with the map data will be lost forever. A new map segment can be created in its place with the function `kpds_create_map`.

If the mapping mode is set to map the data. then this function will not destroy the map. This is because destroying the map when operating in this mode would cause an inconsistency in the interpretation of the value data. `KPDS_MAPPING_MODE` is an attribute which, when set to `KMAPPED`, causes the map data to be unmapped into the value data. For example, if you are operating on a pseudo-colored image, and set mapping mode, the result will be that the value data will appear to be a true-color image.

G.5.6. `kpds_query_map()` — *determine if the map segment exists in a data object.*

Synopsis

```
int kpds_query_map(  
    kobject object)
```

Input Arguments

`object`
data object to be queried.

Returns

TRUE if the map segment exists, FALSE otherwise

Description

This function is used to determine if the map segment exists in a data object. If map segment exists in the specified object, then this function will return TRUE. If the object is invalid, or map data does not exist in the object, then this function will return FALSE.

When the mapping mode is set to map the data, the data object will always appear to not contain a map.

G.6. Mask Functions

- *kpds_copy_mask()* - copy the mask segment from one object to another.
- *kpds_copy_mask_attr()* - copy all mask attributes from one object to another object.
- *kpds_copy_mask_data()* - copy all mask data from one object to another object.
- *kpds_create_mask()* - create a mask segment within a data object.
- *kpds_destroy_mask()* - destroy the mask segment in a data object.
- *kpds_query_mask()* - determine if the mask segment exists in a data object.

G.6.1. *kpds_copy_mask()* — *copy the mask segment from one object to another.*

Synopsis

```
int kpds_copy_mask(  
    kobject object1,  
    kobject object2,  
    int      copy_through_presentation)
```

Input Arguments

`object1`
the object that serves as a source for the mask segment.

`copy_through_presentation`
if set to TRUE, the copy will be performed through the presentation of the source and destination objects. if set to FALSE, the copy will be a direct copy of the physical data.

Output Arguments

`object2`
the object that will serve as a destination for the copy operation.

Returns

TRUE if copy was successful, FALSE otherwise

Description

This function copies all of the attributes and data contained in the mask segment of one object into the mask segment contained in another object. If the mask segment exists in the destination object, then its data will be replaced with the data from the source object. If the mask segment does not exist in the destination object, it will be created.

All mask segment attributes will be copied from one object to the other. This includes presentation attributes, such as position and offset.

The mask data can be optionally copied through the presentations of the two data objects. This implies that any presentation stages which are normally invoked during a `kpds_get_data` on the source object or a `kpds_put_data` call on the destination object will be used for the copy.

Any of the following presentation stages can be invoked on either the source object or the destination object : casting, scaling, normalizing, padding, and interpolating.

The following presentation stages can be invoked on only the source object : mapping, axis assignment, position, and offset. The resulting copy will be mapped through the map and transposed to be in the default axis ordering. The resulting copy will also appear to be shifted by the source position and offset.

These presentation stages are brought into the data pipeline by setting the appropriate presentation attributes on the data objects. Please see the chapter on polymorphic data services in the data services manual for more information on the various presentation attributes.

The following example may help in visualizing the process. The source object has a presentation data type different from the physical data type, and a presentation size different from the physical size. The destination object only has a different presentation and physical data type.

```
source -> interpolating -> casting -|...| -> casting -> destination
```

In the above example, the data resulting from the copy will be interpolated to a new size and cast to a new data type before being copied into the destination object. When being copied into the destination object, the data will be cast yet again before it finally is stored.

If the presentation and physical layers of the destination object are coupled then the destination attributes will be ignored and the source presentation attributes will be propagated to the destination physical layer. The copy, in essence, will be performed only through the source presentation, with the destination physical and presentation layers taking on the characteristics of the source presentation. By default, output objects are not coupled, so this behavior is typical.

The copy need not be performed using the presentation. If the data is not copied through the presentation, the destination data will be a direct copy of the physical source data. The physical size and data type of the mask segment will be reflected in the destination data object from the source to the destination since they describe the physical state of the data.

This function is equivalent to performing successive calls to `kpds_copy_mask_attr` and `kpds_copy_mask_data`.

G.6.2. `kpds_copy_mask_attr()` — *copy all mask attributes from one object to another object.*

Synopsis

```
int kpds_copy_mask_attr(  
    kobject object1,  
    kobject object2)
```

Returns

TRUE if copy was successful, FALSE otherwise

Description

This function copies all the mask attributes from one object to another object.

If the destination object does not contain a mask segment, then this function will create a mask segment, and initialize its size and data type attributes to those in the source object. If the mask data already exists in the destination object, then the presentation attributes will be set to the source object's settings.

The destination's physical attributes will change depending on the coupling of the data objects. If the destination is coupled, then the physical attributes will be changed as well. Any data contained in the destination will be cast, rescaled, or resized to match its new physical attributes.

G.6.3. `kpds_copy_mask_data()` — *copy all mask data from one object to another object.*

Synopsis

```
int kpds_copy_mask_data(  
    kobject object1,  
    kobject object2,  
    int      copy_through_presentation)
```

Input Arguments

`object1`
the object that serves as a source for the data.

`copy_through_presentation`
if set to TRUE, the copy will be performed through the presentation of the source and destination objects. if set to FALSE, the copy will be a direct copy of the physical data.

Output Arguments

`object2`

the object that will serve as a destination for the copy operation.

Returns

TRUE if copy was successful, FALSE otherwise

Description

This function copies all of the data contained in the mask segment of one object into the mask segment contained in another object. If the mask segment exists in the destination object, then its data will be replaced with the data from the source object. If the mask segment does not exist in the destination object, it will be created.

The mask data can be optionally copied through the presentations of the two data objects. This implies that any presentation stages which are normally invoked during a `kpds_get_data` on the source object or a `kpds_put_data` call on the destination object will be used for the copy.

Any of the following presentation stages can be invoked on either the source object or the destination object : casting, scaling, normalizing, padding, and interpolating.

The following presentation stages can be invoked on only the source object : mapping, axis assignment, position, and offset. The resulting copy will be set to the size it appears to be when mapping mode is set to mapped, and will be transposed to be in the default axis ordering. The resulting copy will also appear to be shifted by the source position and offset.

These presentation stages are brought into the data pipeline by setting the appropriate presentation attributes on the data objects. Please see the chapter on polymorphic data services in the data services manual for more information on the various presentation attributes.

The following example may help in visualizing the process. The source object has a presentation data type different from the physical data type, and a presentation size different from the physical size. The destination object only has a different presentation and physical data type.

```
source -> interpolating -> casting -[...] -> casting -> destination
```

In the above example, the data resulting from the copy will be interpolated to a new size and cast to a new data type before being copied into the destination object. When being copied into the destination object, the data will be cast yet again before it finally is stored.

If the presentation and physical layers of the destination object are coupled then the destination attributes will be ignored and the source presentation attributes will be propagated to the destination physical layer. The copy, in essence, will be performed only through the source presentation, with the destination physical and presentation layers taking on the characteristics of the source presentation. By default, output objects are not coupled, so this behavior is typical.

The copy need not be performed using the presentation. If the data is not copied through the presentation, the destination data will be a direct copy of the physical source data. The physical size and data type of the mask segment will be reflected in the destination data object from the source to the

destination since they describe the physical state of the data.

G.6.4. `kpds_create_mask()` — *create a mask segment within a data object.*

Synopsis

```
int kpds_create_mask(  
    kobject object)
```

Input Arguments

`object`
object in which to create the mask segment.

Returns

TRUE if the mask segment was successfully created, FALSE otherwise

Description

This function is used to create a mask segment within a specified data object. The size of the mask segment will be initialized to match any the sizes shared with any other polymorphic segments.

Note that the pad value for this segment is initialized to 1. This is done so that padded data is marked as valid according to the mask segment.

It is considered an error to create a mask segment if the object already contains one.

G.6.5. `kpds_destroy_mask()` — *destroy the mask segment in a data object.*

Synopsis

```
int kpds_destroy_mask(  
    kobject object)
```

Input Arguments

`object`
object from which to remove the mask segment.

Returns

TRUE if the mask segment is successfully destroyed, FALSE otherwise

Description

This function is used to destroy the mask segment contained within an object. Once the mask segment has been destroyed, any data or attributes associated with the mask data will be lost forever. A new mask segment can be created in its place with the function `kpds_create_mask`.

If the mask segment does not exist in the specified object, it is considered to be an error.

G.6.6. `kpds_query_mask()` — *determine if the mask segment exists in a data object.*

Synopsis

```
int kpds_query_mask(  
    kobject object)
```

Input Arguments

`object`
data object to be queried.

Returns

TRUE if the mask segment exists, FALSE otherwise

Description

This function is used to determine if the mask segment exists in a data object. If mask segment exists in the specified object, then this function will return TRUE. If the object is invalid, or mask data does not exist in the object, then this function will return FALSE.

G.7. Time Functions

- `kpds_copy_time()` - copy the time segment from one object to another.
- `kpds_copy_time_attr()` - copy all time attributes from one object to another object.
- `kpds_copy_time_data()` - copy all time data from one object to another object.
- `kpds_create_time()` - create a time segment within a data object.
- `kpds_destroy_time()` - destroy the time segment in a data object.
- `kpds_query_time()` - determine if the time segment exists in a data object.

G.7.1. `kpds_copy_time()` — *copy the time segment from one object to another.*

Synopsis

```
int kpds_copy_time(  
    kobject object,  
    kobject dest)
```

```
kobject object1,  
kobject object2,  
int      copy_through_presentation)
```

Input Arguments

`object1`

the object that serves as a source for the time segment.

`copy_through_presentation`

if set to TRUE, the copy will be performed through the presentation of the source and destination objects. if set to FALSE, the copy will be a direct copy of the physical data.

Output Arguments

`object2`

the object that will serve as a destination for the copy operation.

Returns

TRUE if copy was successful, FALSE otherwise

Description

This function copies all of the attributes and data contained in the time segment of one object into the time segment contained in another object. If the time segment exists in the destination object, then its data will be replaced with the data from the source object. If the time segment does not exist in the destination object, it will be created.

All time segment attributes will be copied from one object to the other. This includes presentation attributes, such as position and offset.

The time data can be optionally copied through the presentations of the two data objects. This implies that any presentation stages which are normally invoked during a `kpds_get_data` on the source object or a `kpds_put_data` call on the destination object will be used for the copy.

Any of the following presentation stages can be invoked on either the source object or the destination object : casting, scaling, normalizing, padding, and interpolating.

The following presentation stages can be invoked on only the source object : position, and offset. The resulting copy will appear to be shifted by the source position and offset.

These presentation stages are brought into the data pipeline by setting the appropriate presentation attributes on the data objects. Please see the chapter on polymorphic data services in the data services manual for more information on the various presentation attributes.

The following example may help in visualizing the process. The source object has a presentation data type different from the physical data type and a presentation size different from the physical size. The destination object only has a different presentation and physical data type.

source -> interpolating -> casting -|...| -> casting -> destination

In the above example, the data resulting from the copy will be interpolated to a new size and cast to a new data type before being copied into the destination object. When being copied into the destination object, the data will be cast yet again before it finally is stored.

If the presentation and physical layers of the destination object are coupled then the destination attributes will be ignored and the source presentation attributes will be propagated to the destination physical layer. The copy, in essence, will be performed only through the source presentation, with the destination physical and presentation layers taking on the characteristics of the source presentation. By default, output objects are not coupled, so this behavior is typical.

The copy need not be performed using the presentation. If the data is not copied through the presentation, the destination data will be a direct copy of the physical source data. The physical size and data type of the time segment will be reflected in the destination data object from the source to the destination since they describe the physical state of the data.

This function is equivalent to performing successive calls to `kpds_copy_time_attr` and `kpds_copy_time_data`.

G.7.2. <code>kpds_copy_time_attr()</code> — <i>copy all time attributes from one object to another object.</i>

Synopsis

```
int kpds_copy_time_attr(  
    kobject object1,  
    kobject object2)
```

Returns

TRUE if copy was successful, FALSE otherwise

Description

This function copies all the time attributes from one object to another object.

If the destination object does not contain a time segment, then this function will create a time segment, and initialize its size and data type attributes to those in the source object. If the time data already exists in the destination object, then the presentation attributes will be set to the source object's settings.

The destination's physical attributes will change depending on the coupling of the data objects. If the destination is coupled, then the physical attributes will be changed as well. Any data contained in the destination will be cast, rescaled, or resized to match its new physical attributes.

G.7.3. `kpds_copy_time_data()` — *copy all time data from one object to another object.*

Synopsis

```
int kpds_copy_time_data(  
    kobject object1,  
    kobject object2,  
    int      copy_through_presentation)
```

Input Arguments

`object1`

the object that serves as a source for the data.

`copy_through_presentation`

if set to TRUE, the copy will be performed through the presentation of the source and destination objects. if set to FALSE, the copy will be a direct copy of the physical data.

Output Arguments

`object2`

the object that will serve as a destination for the copy operation.

Returns

TRUE if copy was successful, FALSE otherwise

Description

This function copies all of the data contained in the time segment of one object into the time segment contained in another object. If the time segment exists in the destination object, then its data will be replaced with the data from the source object. If the time segment does not exist in the destination object, it will be created.

The time data can be optionally copied through the presentations of the two data objects. This implies that any presentation stages which are normally invoked during a `kpds_get_data` on the source object or a `kpds_put_data` call on the destination object will be used for the copy.

Any of the following presentation stages can be invoked on either the source object or the destination object : casting, scaling, normalizing, padding, and interpolating.

The following presentation stages can be invoked on only the source object : position, and offset. The resulting copy will appear to be shifted by the source position and offset.

These presentation stages are brought into the data pipeline by setting the appropriate presentation attributes on the data objects. Please see the chapter on polymorphic data services in the data services manual for more information on the various presentation attributes.

The following example may help in visualizing the process. The source object has a presentation data type different from the physical data type and a presentation size different from the physical size. The

destination object only has a different presentation and physical data type.

source -> interpolating -> casting -[...] -> casting -> destination

In the above example, the data resulting from the copy will be interpolated to a new size and cast to a new data type before being copied into the destination object. When being copied into the destination object, the data will be cast yet again before it finally is stored.

If the presentation and physical layers of the destination object are coupled then the destination attributes will be ignored and the source presentation attributes will be propagated to the destination physical layer. The copy, in essence, will be performed only through the source presentation, with the destination physical and presentation layers taking on the characteristics of the source presentation. By default, output objects are not coupled, so this behavior is typical.

The copy need not be performed using the presentation. If the data is not copied through the presentation, the destination data will be a direct copy of the physical source data. The physical size and data type of the time segment will be reflected in the destination data object from the source to the destination since they describe the physical state of the data.

G.7.4. `kpds_create_time()` — *create a time segment within a data object.*

Synopsis

```
int kpds_create_time(  
    kobject object)
```

Input Arguments

`object`
object in which to create the time segment.

Returns

TRUE if the time segment was successfully created, FALSE otherwise

Description

This function is used to create a time segment within a specified data object. The size of the time segment will be initialized to match any the time size shared with the other polymorphic segments.

It is considered an error to create a time segment if the object already contains one.

G.7.5. `kpds_destroy_time()` — *destroy the time segment in a data object.*

Synopsis

```
int kpds_destroy_time(  
    kobject object)
```

Input Arguments

`object`
object from which to remove the time segment.

Returns

TRUE if the time segment is successfully destroyed, FALSE otherwise

Description

This function is used to destroy the time segment contained within an object. Once the time segment has been destroyed, any data or attributes associated with the time data will be lost forever. A new time segment can be created in its place with the function `kpds_create_time`.

If the time segment does not exist in the specified object, it is considered to be an error.

G.7.6. `kpds_query_time()` — *determine if the time segment exists in a data object.*

Synopsis

```
int kpds_query_time(  
    kobject object)
```

Input Arguments

`object`
data object to be queried.

Returns

TRUE if the time segment exists, FALSE otherwise

Description

This function is used to determine if the time segment exists in a data object. If time segment exists in the specified object, then this function will return TRUE. If the object is invalid, or time data does not exist in the object, then this function will return FALSE.

G.8. Value Functions

- *kpds_copy_value()* - copy the value segment from one object to another.
- *kpds_copy_value_attr()* - copy all value attributes from one object to another object.
- *kpds_copy_value_data()* - copy all value data from one object to another object.
- *kpds_create_value()* - create a value segment within a data object.
- *kpds_destroy_value()* - destroy the value segment in a data object.
- *kpds_query_value()* - determine if the value segment exists in a data object.

G.8.1. *kpds_copy_value()* — *copy the value segment from one object to another.*

Synopsis

```
int kpds_copy_value(  
    kobject object1,  
    kobject object2,  
    int      copy_through_presentation)
```

Input Arguments

object1
the object that serves as a source for the value segment.

copy_through_presentation
if set to TRUE, the copy will be performed through the presentation of the source and destination objects. if set to FALSE, the copy will be a direct copy of the physical data.

Output Arguments

object2
the object that will serve as a destination for the copy operation.

Returns

TRUE if copy was successful, FALSE otherwise

Description

This function copies all of the attributes and data contained in the value segment of one object into the value segment contained in another object. If the value segment exists in the destination object, then its data will be replaced with the data from the source object. If the value segment does not exist in the destination object, it will be created.

All value segment attributes will be copied from one object to the other. This includes presentation attributes, such as position and offset.

The value data can be optionally copied through the presentations of the two data objects. This implies that any presentation stages which are normally invoked during a *kpds_get_data* on the source object or a *kpds_put_data* call on the destination object will be used for the copy.

Any of the following presentation stages can be invoked on either the source object or the destination object : casting, scaling, normalizing, padding, and interpolating.

The following presentation stages can be invoked on only the source object : mapping, axis assignment, position, and offset. The resulting copy will be mapped through the map and transposed to be in the default axis ordering. The resulting copy will also appear to be shifted by the source position and offset.

These presentation stages are brought into the data pipeline by setting the appropriate presentation attributes on the data objects. Please see the chapter on polymorphic data services in the data services manual for more information on the various presentation attributes.

The following example may help in visualizing the process. The source object has mapping mode set to map the data, a presentation data type different from the physical data type, and a presentation size different from the physical size. The destination object only has a different presentation and physical data type.

```
source -> casting -> mapping -|...| -> casting -> destination
```

In the above example, the data resulting from the copy will be cast to a new data type and mapped through the object's map before being copied into the destination object. When being copied into the destination object, the data will be cast yet again before it finally is stored.

If the presentation and physical layers of the destination object are coupled then the destination attributes will be ignored and the source presentation attributes will be propagated to the destination physical layer. The copy, in essence, will be performed only through the source presentation, with the destination physical and presentation layers taking on the characteristics of the source presentation. By default, output objects are not coupled, so this behavior is typical.

The copy need not be performed using the presentation. If the data is not copied through the presentation, the destination data will be a direct copy of the physical source data. The physical size and data type of the value segment will be reflected in the destination data object from the source to the destination since they describe the physical state of the data.

This function is equivalent to performing successive calls to `kpds_copy_value_attr` and `kpds_copy_value_data`.

G.8.2. <code>kpds_copy_value_attr()</code> — <i>copy all value attributes from one object to another object.</i>

Synopsis

```
int kpds_copy_value_attr(  
    kobject sobject,  
    kobject dobject)
```

Input Arguments

`subject`

the object that serves as a source for the attributes.

Output Arguments

`dobject`

the object that serves as a destination for the operation.

Returns

TRUE if copy was successful, FALSE otherwise

Description

This function copies all the value attributes from one object to another object.

If the destination object does not contain a value segment, then this function will create a value segment, and initialize its size and data type attributes to those in the source object. If the value data already exists in the destination object, then the presentation attributes will be set to the source object's settings.

The destination's physical attributes will change depending on the coupling of the data objects. If the destination is coupled, then the physical attributes will be changed as well. Any data contained in the destination will be cast, rescaled, or resized to match its new physical attributes.

G.8.3. `kpds_copy_value_data()` — *copy all value data from one object to another object.*

Synopsis

```
int kpds_copy_value_data(  
    kobject object1,  
    kobject object2,  
    int      copy_through_presentation)
```

Input Arguments

`object1`

the object that serves as a source for the data.

`copy_through_presentation`

if set to TRUE, the copy will be performed through the presentation of the source and destination objects. if set to FALSE, the copy will be a direct copy of the physical data.

Output Arguments

`object2`

the object that will serve as a destination for the copy operation.

Returns

TRUE if copy was successful, FALSE otherwise

Description

This function copies all of the data contained in the value segment of one object into the value segment contained in another object. If the value segment exists in the destination object, then its data will be replaced with the data from the source object. If the value segment does not exist in the destination object, it will be created.

The value data can be optionally copied through the presentations of the two data objects. This implies that any presentation stages which are normally invoked during a `kpds_get_data` on the source object or a `kpds_put_data` call on the destination object will be used for the copy.

Any of the following presentation stages can be invoked on either the source object or the destination object : casting, scaling, normalizing, padding, and interpolating.

The following presentation stages can be invoked on only the source object : mapping, axis assignment, position, and offset. The resulting copy will be mapped through the map and transposed to be in the default axis ordering. The resulting copy will also appear to be shifted by the source position and offset.

These presentation stages are brought into the data pipeline by setting the appropriate presentation attributes on the data objects. Please see the chapter on polymorphic data services in the data services manual for more information on the various presentation attributes.

The following example may help in visualizing the process. The source object has mapping mode set to map the data, a presentation data type different from the physical data type, and a presentation size different from the physical size. The destination object only has a different presentation and physical data type.

source -> casting -> mapping -|...| -> casting -> destination

In the above example, the data resulting from the copy will be cast to a new data type and mapped through the object's map before being copied into the destination object. When being copied into the destination object, the data will be cast yet again before it finally is stored.

If the presentation and physical layers of the destination object are coupled then the destination attributes will be ignored and the source presentation attributes will be propagated to the destination physical layer. The copy, in essence, will be performed only through the source presentation, with the destination physical and presentation layers taking on the characteristics of the source presentation. By default, output objects are not coupled, so this behavior is typical.

The copy need not be performed using the presentation. If the data is not copied through the presentation, the destination data will be a direct copy of the physical source data. The physical size and data type of the value segment will be reflected in the destination data object from the source to the destination since they describe the physical state of the data.

G.8.4. `kpds_create_value()` — *create a value segment within a data object.*

Synopsis

```
int kpds_create_value(  
    kobject object)
```

Input Arguments

object
object in which to create the value segment.

Returns

TRUE if the value segment was successfully created, FALSE otherwise

Description

This function is used to create a value segment within a specified data object. The size of the value segment will be initialized to match any the sizes shared with any other polymorphic segments.

It is considered an error to create a value segment if the object already contains one.

G.8.5. `kpds_destroy_value()` — *destroy the value segment in a data object.*

Synopsis

```
int kpds_destroy_value(  
    kobject object)
```

Input Arguments

`object`
object from which to remove the value segment.

Returns

TRUE if the value segment is successfully destroyed, FALSE otherwise

Description

This function is used to destroy the value segment contained within an object. Once the value segment has been destroyed, any data or attributes associated with the value data will be lost forever. A new value segment can be created in its place with the function `kpds_create_value`.

If the value segment does not exist in the specified object, it is considered to be an error.

G.8.6. `kpds_query_value()` — *determine if the value segment exists in a data object.*

Synopsis

```
int kpds_query_value(  
    kobject object)
```

Input Arguments

`object`
data object to be queried.

Returns

TRUE if the value segment exists, FALSE otherwise

Description

This function is used to determine if the value segment exists in a data object. If value segment exists in the specified object, then this function will return TRUE. If the object is invalid, or value data does not exist in the object, then this function will return FALSE.

This page left intentionally blank

Table of Contents

A. Introduction	2-1
B. The Polymorphic Data Model	2-2
B.1. Value Data	2-4
B.2. Mask	2-5
B.3. Map	2-5
B.4. Location	2-6
B.5. Time	2-7
C. Interaction with the Polymorphic Data Model	2-8
C.1. Presentation of the Data Object	2-8
C.2. Casting	2-9
C.3. Scaling and Normalization	2-9
C.4. Padding and Interpolation	2-10
C.5. Conversion of Complex Data	2-10
C.6. Map Evaluation	2-11
C.7. Mask Evaluation	2-11
C.8. Axis Assignment	2-11
C.9. Data Ranging	2-12
C.10. Reference Objects	2-12
C.11. Auto Incrementing	2-13
D. The Application Programming Interface (API)	2-14
E. Polymorphic Primitives	2-17
E.1. Value Primitives	2-17
E.2. Mask Primitives	2-19
E.3. Map Primitives	2-22
E.4. Location Primitives	2-24
E.4.1. Creating Location	2-26
E.4.2. Location Primitives	2-26
E.4.3. Presentation of Location Data	2-30
E.5. Time Primitives	2-31
F. Attributes Defined by the Polymorphic Data Model	2-31
F.1. Global Attributes	2-33
F.2. Value Segment Attributes	2-38
F.3. Mask Segment Attributes	2-44
F.4. Map Segment Attributes	2-49
F.5. Location Segment Attributes	2-55
F.6. Time Segment Attributes	2-61
G. Functions Provided by Polymorphic Data Services	2-66
G.1. Object Management	2-66
G.1.1. <code>kpds_open_input_object()</code> — <i>open an input object for reading</i>	2-67
G.1.2. <code>kpds_open_output_object()</code> — <i>open an output object for writing</i>	2-68
G.1.3. <code>kpds_create_object()</code> — <i>create a temporary data object.</i>	2-69
G.1.4. <code>kpds_create_object_attr()</code> — <i>create an attribute associated the data object.</i>	2-69
G.1.5. <code>kpds_destroy_object_attr()</code> — <i>destroy an attribute associated with the data object.</i>	2-71
G.1.6. <code>kpds_open_object()</code> — <i>create an object associated with an input or output transport.</i>	2-72
G.1.7. <code>kpds_close_object()</code> — <i>close an open data object.</i>	2-74
G.1.8. <code>kpds_reference_object()</code> — <i>create a reference of a data object.</i>	2-74
G.1.9. <code>kpds_copy_object()</code> — <i>copy all data and attributes from one object to another.</i>	2-75

G.1.10. <code>kpds_copy_remaining_data()</code> — <i>copy remaining data from source to destination</i>	2-76
G.1.11. <code>kpds_copy_object_attr()</code> — <i>copy all presentation attributes from one data object to another.</i>	2-77
G.1.12. <code>kpds_copy_object_data()</code> — <i>copy all data from one object to another object.</i>	2-77
G.1.13. <code>kpds_sync_object()</code> — <i>synchronize physical and presentation layers of a data object.</i>	2-78
G.2. Data Functions	2-79
G.2.1. <code>kpds_get_data()</code> — <i>retrieve data from a data object.</i>	2-79
G.2.2. <code>kpds_put_data()</code> — <i>store data in a data object.</i>	2-82
G.3. Attribute Functions	2-84
G.3.1. <code>kpds_copy_attribute()</code> — <i>copy an attribute from one object to another</i>	2-84
G.3.2. <code>kpds_copy_attributes()</code> — <i>copy multiple attributes from one object to another.</i>	2-85
G.3.3. <code>kpds_get_attribute()</code> — <i>get the value of an attribute from a data object</i>	2-86
G.3.4. <code>kpds_get_attributes()</code> — <i>get the values of multiple attributes from a data object</i>	2-87
G.3.5. <code>kpds_match_attribute()</code> — <i>returns TRUE if the same attribute in two objects match.</i>	2-89
G.3.6. <code>kpds_match_attributes()</code> — <i>returns true if the list of segment attributes in two objects match.</i>	2-90
G.3.7. <code>kpds_print_attribute()</code> — <i>print the value of an attribute from a data object.</i>	2-91
G.3.8. <code>kpds_query_attribute()</code> — <i>get information about an attribute</i>	2-91
G.3.9. <code>kpds_set_attribute()</code> — <i>set the values of an attribute in a data object</i>	2-93
G.3.10. <code>kpds_set_attributes()</code> — <i>set the values of multiple attributes in a data object.</i>	2-94
G.4. Location Functions	2-95
G.4.1. <code>kpds_copy_location()</code> — <i>copy the location segment from one object to another.</i>	2-96
G.4.2. <code>kpds_copy_location_attr()</code> — <i>copy all location attributes from one object to another object.</i>	2-97
G.4.3. <code>kpds_copy_location_data()</code> — <i>copy all location data from one object to another object.</i>	2-98
G.4.4. <code>kpds_create_location()</code> — <i>create a location segment within a data object.</i>	2-100
G.4.5. <code>kpds_destroy_location()</code> — <i>destroy the location segment in a data object.</i>	2-100
G.4.6. <code>kpds_query_location()</code> — <i>determine if the location segment exists in a data object.</i>	2-101
G.5. Map Functions	2-101
G.5.1. <code>kpds_copy_map()</code> — <i>copy the map segment from one object to another.</i>	2-102
G.5.2. <code>kpds_copy_map_attr()</code> — <i>copy all map attributes from one object to another object.</i>	2-103
G.5.3. <code>kpds_copy_map_data()</code> — <i>copy all map data from one object to another object.</i>	2-105
G.5.4. <code>kpds_create_map()</code> — <i>create a map segment within a data object.</i>	2-106
G.5.5. <code>kpds_destroy_map()</code> — <i>destroy the map segment in a data object.</i>	2-107
G.5.6. <code>kpds_query_map()</code> — <i>determine if the map segment exists in a data object.</i>	2-107
G.6. Mask Functions	2-108
G.6.1. <code>kpds_copy_mask()</code> — <i>copy the mask segment from one object to another.</i>	2-108
G.6.2. <code>kpds_copy_mask_attr()</code> — <i>copy all mask attributes from one object to another object.</i>	2-110
G.6.3. <code>kpds_copy_mask_data()</code> — <i>copy all mask data from one object to another object.</i>	2-110
G.6.4. <code>kpds_create_mask()</code> — <i>create a mask segment within a data object.</i>	2-112
G.6.5. <code>kpds_destroy_mask()</code> — <i>destroy the mask segment in a data object.</i>	2-112
G.6.6. <code>kpds_query_mask()</code> — <i>determine if the mask segment exists in a data object.</i>	2-113
G.7. Time Functions	2-113
G.7.1. <code>kpds_copy_time()</code> — <i>copy the time segment from one object to another.</i>	2-113
G.7.2. <code>kpds_copy_time_attr()</code> — <i>copy all time attributes from one object to another object.</i>	2-115
G.7.3. <code>kpds_copy_time_data()</code> — <i>copy all time data from one object to another object.</i>	2-116
G.7.4. <code>kpds_create_time()</code> — <i>create a time segment within a data object.</i>	2-117
G.7.5. <code>kpds_destroy_time()</code> — <i>destroy the time segment in a data object.</i>	2-118

G.7.6. <code>kpds_query_time()</code> — <i>determine if the time segment exists in a data object.</i>2-118
G.8. Value Functions2-119
G.8.1. <code>kpds_copy_value()</code> — <i>copy the value segment from one object to another.</i>2-119
G.8.2. <code>kpds_copy_value_attr()</code> — <i>copy all value attributes from one object to another object.</i>2-120
G.8.3. <code>kpds_copy_value_data()</code> — <i>copy all value data from one object to another object.</i>2-122
G.8.4. <code>kpds_create_value()</code> — <i>create a value segment within a data object.</i>2-123
G.8.5. <code>kpds_destroy_value()</code> — <i>destroy the value segment in a data object.</i>2-124
G.8.6. <code>kpds_query_value()</code> — <i>determine if the value segment exists in a data object.</i>2-124

This page left intentionally blank

Chapter 3

Geometry Data Services

Chapter 3 - Geometry Data Services

A. Geometry Data Services

A.1. Introduction

Geometry services provides a general mechanism for the storage and retrieval of geometry data. In terms of data visualization, geometry data describes the shape and position of structures in space. Geometry data therefore consists primarily of spatial information in the form of explicit location data. Furthermore, geometry data may also include data which does not have any direct bearing on the spatial description of the structure; color, normal, and texture data are examples. In addition to geometry data, volumetric data is also supported since it too describes structures in space. Finally, geometry services also allows you to store attributes such as a name or opacity value with the data.

With geometry services, geometry data is stored and retrieved from a *data structure*. The structure fields describe the *geometry data model*. In contrast with the other VisiQuest data services, geometry data services provides you with direct access to the geometry data structure. Data is stored in a geometry object by assigning the data pointer to a field of a structure. Attributes are stored by assigning the appropriate fields of a structure.

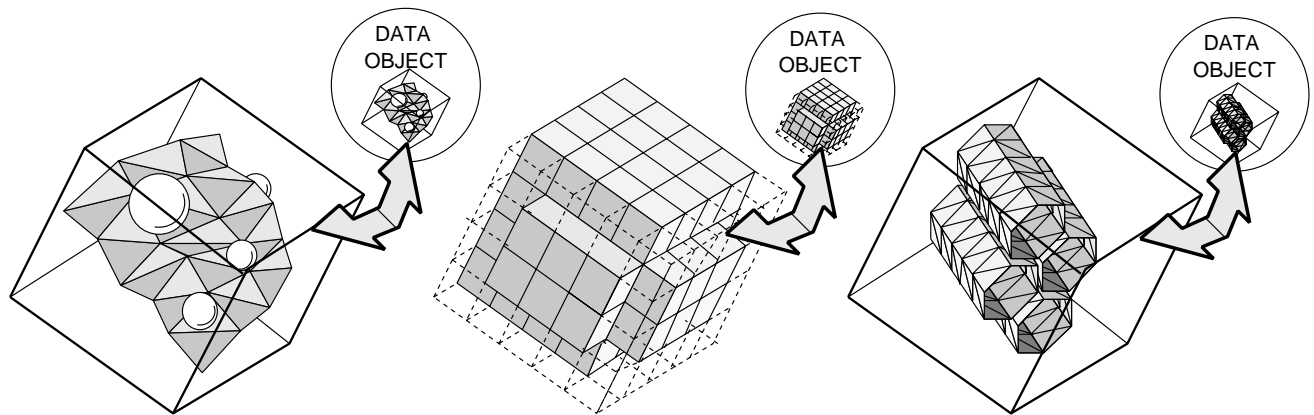


Figure 1: Geometry Services supports the storage and retrieval of geometric data. Geometric data is constructed and stored in a data object. This data can later be retrieved for processing or rendering. Geometry services supports geometric data as well as volumetric data.

An image is produced from geometry through a process called *rendering*. Since a generic geometry service that directly encompassed the needs of many different types of renderers would have too large a scope, geometry services does not attempt to support renderer-specific information. The goal of geometry services is simply to provide a programming framework for processing geometry that separates the creation and storage of geom-

entry data from the manipulation and rendering of geometry.

A.1.1. The Geometry Data Model

The geometry data model is centered around the concept of a **primitive list**. Each geometry object contains a single primitive list. Geometry **primitives** can be added to or removed from this primitive list. A geometry primitive is defined simply to be a grouping of data with a specific geometric interpretation. For example, a polyline or a triangle strip are considered to be geometry primitives. Hierarchical geometry structures may be created by placing one geometry object on the primitive list of another object.

In addition to geometry primitives, a geometry object may contain a number of geometry **attributes**. These attributes are represented by fields within the geometry object structure. Some of these attributes describe global characteristics of the data and apply to all the primitives in the object as a collective group. Examples of these *object-level* attributes are the bounding box or the object color. Primitives also have attributes. These are represented by the structure fields within a primitive structure. Examples of these *primitive level* attributes are the number of vertices in a primitive, or the line type of a primitive.

A.1.1.1. Geometry Primitives

Geometry primitives in a basic sense can be thought of as an aggregation of different data components, such as *location* data and *color* data, that represent a geometric construct. Other data components that are encountered are *normals*, *texture coordinates*, and *radii*. There are many types of geometry primitives, ranging from surface primitives, such as **connected triangles**, to annotation primitives, such as **text**, to volumetric primitives, such as an **octmesh**.

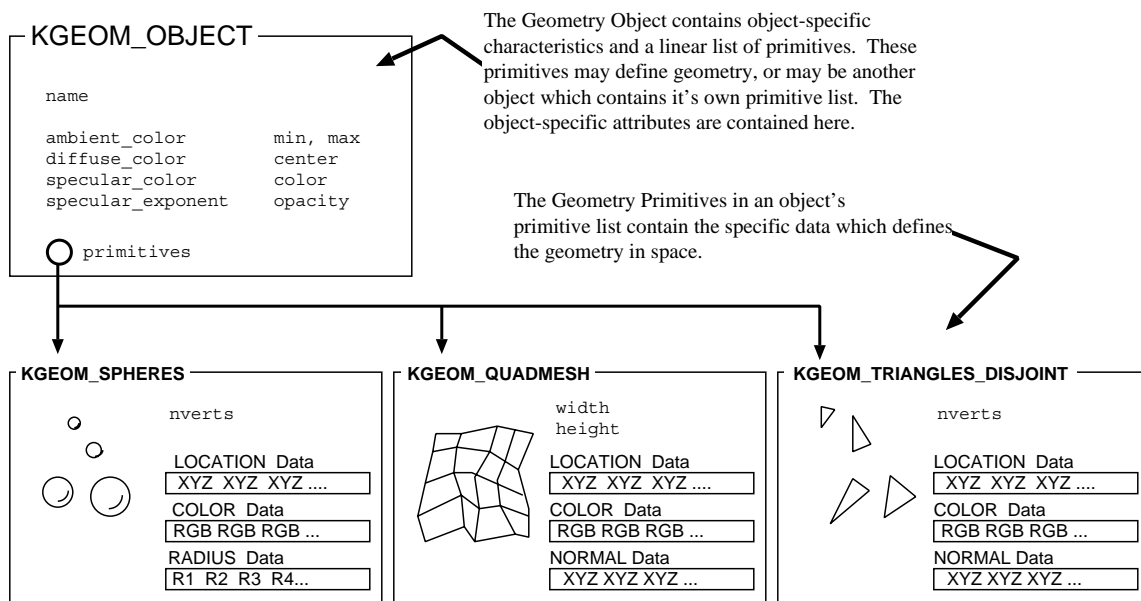


Figure 2: An overview of the Geometry Data Model. A geometry object contains a **primitive list**. Geometric primitives are stored and retrieved from this list. Each geometric primitive is an aggregate of different types of data; for example, a spheres primitive consists of location data, color data, and radius data. Note that a single spheres primitive contains multiple spheres.

Primitives can be combined and stored in any combination within a single object. For example, a **connected polyline** can be combined with a **list of spheres** and a **list of directed points** and placed into a single data object. These primitives are all added to the geometry object's primitive list.

A.1.1.2. VisiQuest Geometry Format

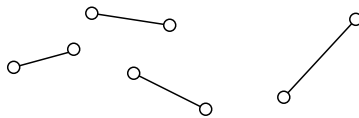
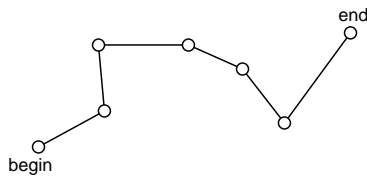
A new file format was created for the storage of geometry objects. This new *VisiQuest Geometry* format is simply a reflection of the geometry object stored in a file. It can be identified by the filename extension "kgm," or by the first 5 bytes, which will spell "kgeom."

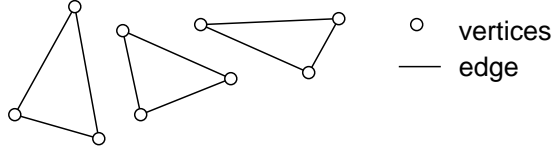
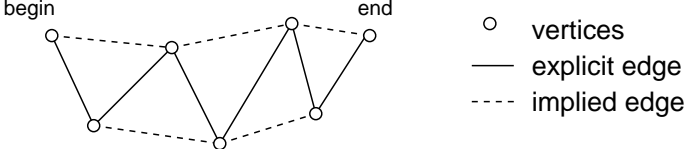
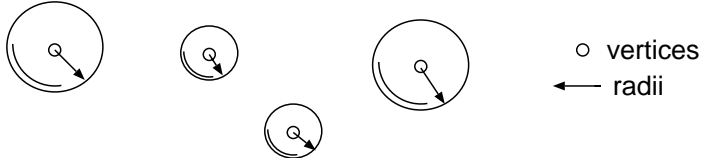
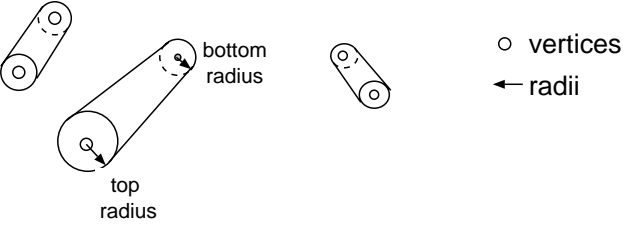
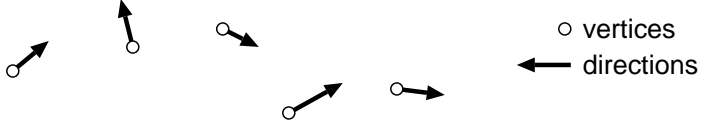
A.2. Overview of Geometry Service Primitives

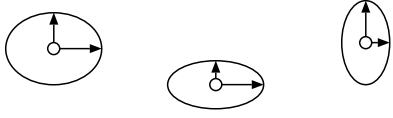
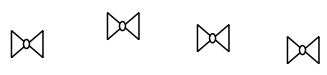
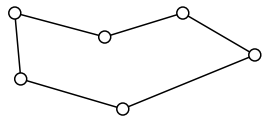
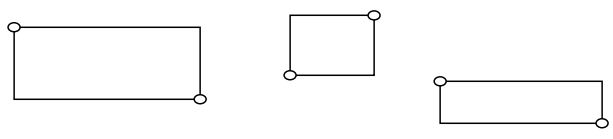
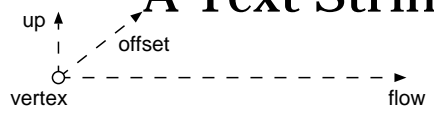
Geometry services supports a number of different geometric primitives: quadmesh primitives, octmesh primitives, two- and three-dimensional texture primitives, as well as other informational primitives.

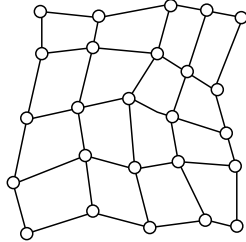
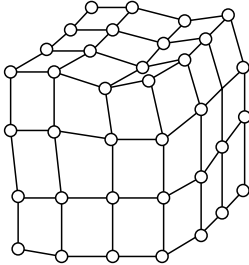

For the purposes of the following descriptions, the term *vertex* is used to describe a point in space-an (x,y,z) point. The location data, which defines the object in space, consists of a series of these points, or vertices. The term *line* is defined to be any vector between two connected vertices. The term *face* is the surface defined by three or more connected vertices that is bounded by the edges connecting those vertices. Colors and normals may exist either per vertex, per line, or per face, while other data, such as texture coords, always exist per vertex.

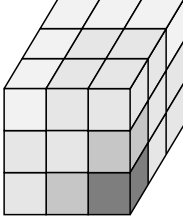
The following table contains descriptions of the geometry primitives provided by geometry services.

Geometry Primitives	
Name	Description and Diagram
KGEOM_POLYLINE_DISJOINT	<p>A disjoint polyline uses N vertices, where N is greater than 1, to construct N/2 lines. Every two vertices represent the end points of a line segment.</p> <div style="text-align: center;">  </div> <p style="text-align: right;">○ vertices</p>
KGEOM_POLYLINE_CONNECTED	<p>A connected polyline uses N vertices, where N is greater than 1, to construct N-1 lines. A line segment connects each adjacent vertex. The first and the last vertices indicate the beginning and the end of the line.</p> <div style="text-align: center;">  </div> <p style="text-align: right;">○ vertices</p>

Geometry Primitives	
Name	Description and Diagram
KGEOM_TRIANGLES_DISJOINT	<p>A disjoint polytriangle uses N vertices, where N is greater than 2, to construct $N/3$ triangles. Every three vertices represent a single triangle.</p> 
KGEOM_TRIANGLES_CONNECTED	<p>A connected polytriangle uses N vertices, where N is greater than 2, to construct $N-2$ triangles. The first and last vertices indicate the beginning and the end of the polytriangle.</p> 
KGEOM_SPHERES	<p>A list of spheres uses N vertices to represent the centers of N spheres. Each sphere has a corresponding radius to indicate its size.</p> 
KGEOM_CYLINDERS	<p>A list of disjoint cylinders uses N vertices to construct $N/2$ cylinders. A cylinder has top radius and bottom radius. If one or the other radii is zero, the the cylinder becomes a cone.</p> 
KGEOM_DIRECTED_POINTS	<p>A list of disjoint discrete points with normals.</p> 

Geometry Primitives	
Name	Description and Diagram
KGEOM_ELLIPSES	<p>A list of ellipses uses N vertices to represent the centers of N ellipses. Each ellipses has a corresponding major and minor radius to indicate its size.</p> <div style="display: flex; align-items: center; justify-content: center;">  <div style="margin-left: 20px;"> <p>○ vertices ↑ minor axis → major axis</p> </div> </div>
KGEOM_MARKERS	<p>A list of disjoint annotation markers of fixed size. For example, a cross hatch or a diamond are markers. These can be thought of as 1D primitives in space.</p> <div style="display: flex; align-items: center; justify-content: center;">  <div style="margin-left: 20px;"> <p>○ vertices</p> </div> </div>
KGEOM_POLYGON	<p>A single two-dimensional polygon defined to be line segments connecting N vertices together. The connectivity of vertices is implied by the vertex order.</p> <div style="display: flex; align-items: center; justify-content: center;">  <div style="margin-left: 20px;"> <p>○ vertices</p> </div> </div>
KGEOM_RECTANGLES	<p>A list of rectangles uses N vertices to construct N/2 rectangles. Every two points define the opposing corners of a rectangle.</p> <div style="display: flex; align-items: center; justify-content: center;">  <div style="margin-left: 20px;"> <p>○ vertices</p> </div> </div>
KGEOM_TEXT	<p>A list of disjoint text strings. Each text string has a vertex describing its origin, and three vectors describing an up direction from the vertex, a flow direction from the vertex, and an offset from the vertex.</p> <div style="text-align: center;"> <h2 style="margin: 0;">A Text String</h2>  </div>

Geometry Primitives	
Name	Description and Diagram
KGEOM_QUADMESH	<p>A quadmesh is a surface formed by a collection of adjacent quadrilaterals. The adjacency, or connectivity, between these quadrilaterals is implied by the two parametric dimensions of <i>width</i> and <i>height</i>. Explicit location vertices are provided to place the quadmesh in space.</p> <div style="text-align: center;">  </div> <p style="text-align: right;">○ vertices</p>
KGEOM_OCTMESH	<p>An octmesh is a volume formed by a collection of adjacent hexahedra. The adjacency, or connectivity between these hexahedra is implied by the three parametric dimensions of <i>width</i>, <i>height</i>, and <i>depth</i>. Explicit locations of vertices are provided to place the quadmesh in space. The number of locations needed to place the mesh in space is dependent on the meshtype.</p> <div style="text-align: center;">  </div> <p style="text-align: right;">○ vertices</p>
KGEOM_TEXTURE2D	<p>A two-dimensional texture is a two-dimensional array of values vectors or colors. The 2d texture primitives are not true geometry primitives, but are accessed through geometry services for ease of programming. The connectivity between colors is implied by their order across the two parametric dimensions of <i>width</i> and <i>height</i>.</p> <div style="text-align: center;">  </div>

Geometry Primitives	
Name	Description and Diagram
KGEOM_TEXTURE3D	<p>A three-dimensional texture is a three-dimensional array of values or colors. The 3d texture primitives are not true geometry primitives, but are accessed through geometry services for ease of programming. The connectivity between colors is implied by their order across the three parametric dimensions of <i>width</i>, <i>height</i>, and <i>depth</i>.</p> 
KGEOM_OBJECT	<p>An object primitive is used when placing one geometry object onto the primitive list of another. This allows the construction of geometric hierarchies. Circular dependencies where two geometry objects are each on the primitive list of the other are not supported.</p>
KGEOM_OBJECT_XFORM	<p>An object transform primitive can be used to specify a transformation directed at some specific, named object. This is would be used, for example, to externally control the transformation of an object external to a rendering program.</p>
KGEOM_VIEW_XFORM	<p>A view transform primitive can be used to specify a viewing transformation. This is would be used, for example, to externally control the viewing position external to a rendering program.</p>
KGEOM_CAMERA_ORIENTATION	<p>A camera orientation primitive can be used to specify the specific orientation and characteristics of a camera. This is would be used, for example, to externally control a camera position external to a rendering program.</p>
KGEOM_VR_EVENT	<p>A vr event primitive is used to send vr specific information from a vr device routine to the rendering routine. It contains information on the current state of the vr device, such as direction of movement, are there any "buttons" being pressed, and so forth.</p>

A.3. The Application Programming Interface (API)

The application programming interface to geometry services consists primarily of constructor and destructor functions for creating and destroying geometry objects and geometry primitives. Functions are also available for writing a geometry object to a file and reading it back.

A.3.1. Geometry Object Functions

A geometry object is represented by a pointer to a `kgeom_object` structure. A geometry object can be instantiated by constructing a new empty object, or by reading in an existing object from a file or other

transport. A new empty object is constructed by the function `kgeom_new_object()` and an existing data object can be read from a file using the function `kgeom_read()`. Once you have a structure filled out, you can write it to a file using the `kgeom_write()` function. And finally, once you are done with the object, you can destroy it and free its resources using the `kgeom_blast_object()` function.

The following examples illustrate the use of these functions:

```
kgeom_object *obj;

/* read in some geometry from a file */
obj = kgeom_read("geometry.geom");

/* do something interesting with the geometry */

/* free up the geometry object when we're done with it */
kgeom_blast_object(obj);
```

Alternatively, if we wanted to create some new geometry and write it to a file, we could use the following code.

```
kgeom_object *obj;

/* create a new geometry object */
obj = kgeom_new_object();

/* create some geometry to go in the object here */

/* once we have the geometry, we can write it out */
kgeom_write(obj, "new_geometry.geom");

/* and we can free it up now that we're done writing it */
kgeom_blast_object(obj);
```

The attributes of the geometry object are stored as the fields of the geometry object structure. These fields are publically accessible; an attribute of a geometry object is set simply by assigning the appropriate structure field. For instance, the following code can be used to set the opacity and name attribute of a geometry object :

```
obj->opacity = 0.5;
obj->name = kstrdup("Isosurface");
```

Note that any string attributes should be duplicated when assigned to the geometry object. In general, geometry services will assume that any data or attributes which are pointed to by the geometry object can be safely freed when the `kgeom_blast_object()` function is called.

A.3.2. Geometry Primitive Functions

A geometry primitive is represented by a pointer to a `kgeom_primitive` structure. As with a geometry object, a geometry primitive can be instantiated either explicitly with the `kgeom_new_primitive()` function, or implicitly when a geometry object is read in from a file. The `kgeom_new_primitive` function takes the type of primitive as its single argument. Each geometry primitive actually has its own distinct structure; the geometry primitive structure is simply a union of all the different primitive structures.

Since the fields of a geometry primitive vary from type to type, the type of primitive being used must be specified in order to access the components of the primitive structure. For example, to access a spheres primitive,

the following syntax should be used :

```
kgeom_primitive *prim;

prim = kgeom_new_primitive(KGEOM_SPHERES);

/* specify that we want 45 spheres */

prim->spheres.nverts = 45;
```

Alternatively, this can be done by casting the structure to its specific primitive type as follows :

```
kgeom_spheres *s;
s = (kgeom_spheres *) kgeom_new_primitive(KGEOM_SPHERES);
/* specify that we want 45 spheres */
s->nverts = 45;
```

Primitives can be cast to any type at any time, but you should be sure that you cast back to the general `kgeom_primitive` type before passing a specific primitive into any functions which take a primitive argument.

A.3.3. Primitive List Functions

Geometry primitives must be stored on the primitive list of an object in order to be written to a file. Once a new primitive has been created and its data assigned, it can be added to the primitive list of an object using the function `kgeom_add_primitive()`. A corresponding function `kgeom_remove_primitive()` allows you to remove a primitive from the primitive list.

The number of primitives which have been added to the primitive list of an object is returned by the function `kgeom_number_primitives()`. A primitive at a specific position in the primitive list can be retrieved using the function `kgeom_get_primitive()`. Note that another geometry object contained on the primitive list of this object will be considered to be a single primitive. The entire subobject will be returned by the `kgeom_get_primitive` function and the primitives contained in the subobject will not be considered when counting the number of primitives in the given object.

These primitive list functions should be the only means used for accessing the primitives on a primitive list. Even though the primitive list of the geometry object could be accessed directly from the fields of the geometry object, it should be considered to be private.

A.3.4. Primitives and Data Vectors

Geometry services provides some flexibility in the organization of the data that is stored for a geometry primitive. The actual amount of data contained in any given primitive for a particular component is dependent on a number of different attributes, as well as the primitive itself. This section will illustrate the attributes that dictate the amount of data present. Once you have a general understanding of these attributes, the specifics of determining how much data is present for a particular primitive will follow.

Note that there are many object-level attributes which will affect the amount of data present in a given primitive. The object whose primitive list contains a given primitive is generally assumed to dictate the object-level

attributes to that primitive. This also implies that all the primitives on the primitive list of an object will share the same attributes. It is not possible to have two line primitives with different layouts on the same primitive list, for instance.

It is also important to note the following: **Data are always organized into a linear array of data vertices.** Even multi-dimensional data, such as quadmesh data, will be handled as a linear array of data. The values composing each vertex are always the leading index in this array. For example a series of (x,y,z) points will be arranged $xyzxyzxyzxyz$ in the linear array.

A.3.4.1. Location Data

Location data consists of a series of vertices that position a primitive in space. Location data is required by nearly all primitives. Location data is always of type `float`.

The number of location vertices in any given primitive is determined by the `nverts` primitive attribute. The only exceptions to this are the mesh and texture primitives, which have their own width, height, and depth attributes. The size of each location vertex is determined by the `location_dim` object attribute. This attribute is by default set to 3 to indicate (x,y,z) vertices.

A.3.4.2. Color Data

Color data is used to provide color to a primitive. Color data is optional. Color data is always of type `float`.

The number of color vectors will be a function of the `nverts` primitive attribute, but may also vary depending on the `layout` object attribute. For example, a disjoint polyline primitive will have 1 color per vertex for a `KPER_VERTEX` layout, but will have 1 color per line for a `KPER_LINE` primitive.

Color vectors generally contain three floats for storing red, green, and blue intensities, although if the object-level attribute `has_alpha` is set to `TRUE`, the color vector will contain an extra float which indicates opacity. These numbers should range from 0.0 to 1.0, with 1.0 implying maximum intensity. If an alpha component is present, it also should range from 0.0 to 1.0, with 0.0 implying that the primitive is totally transparent, and 1.0 implying that the primitive is totally opaque.

A.3.4.3. Normal Data

Normal data consists of a series of vectors that give extra directional and orientation information to a geometry primitive. Normal data is optional. Normal data is always of type `float`.

The number of normal vectors will be a function of the `nverts` primitive attribute, but may also vary depending on the `layout` object attribute. For example, a disjoint triangles primitive will have 1 normal per vertex for a `KPER_VERTEX` layout, but will have 1 normal per triangle for a `KPER_FACE` primitive. The size of each location vertex is determined by the `location_dim` object attribute. This attribute is by default set to 3

to indicate that the normals exist in (x,y,z) space.

A.3.4.4. Radius Data

Radius data consists of a series of values that are interpreted to be the radii of a list of spheres. Radius data is optional. Radius data is of type `float`. The number of radii in any given primitive is determined by the `nverts` primitive attribute.

A.3.4.5. Texture Coordinate Data

Texture coordinate data consists of a series of coordinates that are used to index into a separate texture map. Texture coordinate data is optional. Texture coordinate data will be of type `float`.

The number of texture coordinates in any given primitive is determined by the `nverts` primitive attribute. The only exceptions to this are the mesh and texture primitives, which have their own width, height, and depth attributes. The size of each texture coordinate is determined by the `texture_coord_dim` object attribute. This attribute is by default set to 2 to indicate (u,v) texture coordinates which index into a 2D texture map.

A.3.4.6. Text Data

Text data consists of an array of text strings. Text data may be optional depending on the primitive. The number of text strings in a text primitive is determined by the `nverts` primitive attribute. Each text string is always a NULL terminated `char*`.

A.3.5. Examples

The examples given below illustrate the use of geometry services for reading existing geometry data from a data file, as well as the use of geometry services in storing data to a new file.

A.3.5.1. Reading Geometry Data

The following example presents a simple model of how a program should open up a data object and read the contents of the primitive list using geometry services.

The first step is to declare the necessary variables and read in the geometry object. The geometry object is read with the `kgeom_read()` call.

```
kgeom_object *obj;
int i, number, nverts;
float *locs = NULL;
float *cols = NULL;
float *rads = NULL;

obj = kgeom_read("input.file");
```

The next step is to determine if there is geometry contained within this data object. We will use the `kgeom_number_primitives()` function to determine whether or not the data object contains any primitives.

```
number = kgeom_number_primitives(obj);
```

```

if (number == 0)
{
    kprintf("No geometry in here!");
    kgeom_blast_object(obj);
    kexit(KEXIT_SUCCESS);
}

```

Next, we will loop through the primitive list and process all geometry primitives that we understand. For each geometry primitive, we will examine the data as well as the number of vertices contained in the data. In this example, we will cast the more general `kgeom_primitive` data structure to more specific primitive structures. Once all the primitives are processed, we will free the geometry object using the `kgeom_blast_object()`. For the sake of brevity, we will only consider spheres and disjoint polylines. Also, after retrieving the data, it is likely that it would be processed in some fashion. However, here the data is just examined for the sake of illustration.

```

for (i = 0; i < number; i++)
{
    kgeom_primitive *primitive = kgeom_get_primitive(obj, i);

    switch (primitive->type)
    {
        case KGEOM_SPHERES :
        {
            kgeom_spheres *spheres = (kgeom_spheres *) primitive;

            nverts = spheres->nverts;
            locs = spheres->locs;
            cols = spheres->cols;
            rads = spheres->rads;

            /* presumably something interesting would be done here */

        } break;

        case KGEOM_POLYLINE_DIS :
        {
            kgeom_polyline *lines = (kgeom_polyline *) primitive;

            nverts = lines->nverts;
            locs = lines->locs;
            cols = lines->cols;

            /* presumably something interesting would be done here */

        } break;

        default:
            kprintf("Sample code doesn't recognize that primitive");
    }
}

/* this will free the geometry object and all of its data */
kgeom_blast_object(obj);

```

It is important to be aware that the data pointers within each geometry primitive will be freed when the geometry object is blasted. To keep the data in memory beyond this, we must either duplicate the data or assign the

data pointers in the primitive to `NULL` before blasting the object.

A.3.5.2. Writing Geometry Data

The example below presents a simple model of how a program should output geometry into a data file using geometry services.

The first step is to declare the necessary variables and open the output data object. The new geometry object is allocated with the `kgeom_new_object()` call.

```
kobject *obj;
kgeom_primitive *primitive;

obj = kgeom_new_object();
```

At this time, we may want to set some interesting attributes on the data object. For example, we may wish to assign the name of the geometry object.

```
obj->name = kstrdup("cool dataset");
```

For the most part, the default object attribute values will be sufficient. However, if we want to set any specific attributes that will determine the amount of data in an object, we should do so now. For this example, we will set the layout attribute to dictate that we wish to only have `PER_FACE` data contained in this geometry object.

```
obj->layout = KPER_FACE;
```

We can now begin to create our primitives and add them to the data object. Some components of a geometry primitive are optional. If we have no data to put for these components, we may leave them pointing to `NULL`. After we are done adding all the primitives, we then write the object with the `kgeom_write_object()` function and free it using the `kgeom_blast_object()` function.

```
/*-- add the first primitive -- sphere list with 10 spheres --*/
primitive = kgeom_new_primitive(KGEOM_SPHERES);
primitive->spheres.nverts = 10;
primitive->spheres.locs =
    cool_function_to_get_sphere_locations();
primitive->spheres.rads =
    cool_function_to_get_sphere_radii();
kgeom_add_primitive(obj, primitive);

/*-- add the second primitive
    -- disjoint line with 3 line segments --*/
primitive = kgeom_new_primitive(KGEOM_POLYLINE_DIS);
primitive->polyline.nverts = 6;
primitive->polyline.locs =
    cool_function_to_get_line_locations();
kgeom_add_primitive(obj, primitive);

/*-- add the third primitive
    -- disjoint triangles with 4 triangles --*/
primitive = kgeom_new_primitive(KGEOM_TRIANGLES_DIS);
primitive->triangles.nverts = 12;
primitive->triangles.locs =
    cool_function_to_get_line_locations();
primitive->triangles.norms =
    cool_function_to_get_calculate_norms();
kgeom_add_primitive(obj, primitive);
```

```
kgeom_write_object(obj, "output.file");
kgeom_blast_object(obj);
```

Note that we did not need to free any of the primitives; they will all be freed as part of the geometry object in the `kgeom_blast_object()` call.

Object-Level Attributes		
Attribute and Default	Legal Values	Definition
<pre>ambient_color float ambient_color[3] [.25 .25 .25]</pre>	<pre>0.0 < 1.0</pre>	<p>An object's ambient reflectance is combined source, and is equally scattered throughout a scene. These values indicate the amount of reflected incoming ambient light in each of the R, G, and B components. The value will range from 0.0 to 1.0 where 0.0 implies complete absorption of all ambient light.</p> <p>Persistence: stored</p>
<pre>center float center[3]</pre>		<p>This attribute is the center point of the extent of the geometry contained in this object. The attribute <code>has_center</code> must also be set to TRUE to indicate that there is a center point present.</p> <p>Persistence: stored</p>
<pre>color float color[3] [1.0 1.0 1.0]</pre>	<pre>0.0 < 1.0</pre>	<p>This attribute is an RGB object color which will dictate the object color in the absence of any primitive color. The attribute <code>has_color</code> must also be set to TRUE to indicate that there is an object level color present.</p> <p>Persistence: stored</p>
<pre>diffuse_color float diffuse_color[3] [.5 .5 .5]</pre>	<pre>0.0 < 1.0</pre>	<p>An object's diffuse reflectance is combined 0.0 to 1.0 where 0.0 implies complete absorption of all diffuse light.</p> <p>Persistence: stored</p>
<pre>has_alpha unsigned char has_alpha FALSE</pre>	<pre>TRUE FALSE</pre>	<p>This attribute indicates that opacity values are present in this object. This will imply that all primitive color vectors are of size 4, and that the <code>opacity</code> object attribute should be interpreted.</p> <p>Persistence: stored</p>

Object-Level Attributes		
Attribute and Default	Legal Values	Definition
has_bounding_box unsigned char has_bounding_box FALSE	TRUE FALSE	This attribute indicates that a bounding box is present in the min and max attributes. The min and max attributes define opposing corners of the bounding box. Persistence: stored
has_center unsigned char has_center FALSE	TRUE FALSE	This attribute indicates that a center is present in the center attribute of this object. Persistence: stored
has_color unsigned char has_color FALSE	TRUE FALSE	This attribute indicates that a color is present in the color attribute of this object. Persistence: stored
has_matrix unsigned char has_matrix FALSE	TRUE FALSE	This attribute indicates that a matrix is present in the matrix attribute of this object. Persistence: stored
layout int layout NULL	KPER_VERTEX KPER_LINE KPER_FACE KPER_CELL	The primitives in the object can have per vertex, per face, or per line data. This attribute specifies if color and normal vectors are associated with a vertex, face or edge. Persistence: stored
location_dim int location_dim 3	> 1	This attribute specifies the dimensionality of the location vertices. A dimension of 2 would imply a 2D space with locations specified in x and y. A dimension size of 3 would imply a 3D space with locations specified in x, y, and z. Persistence: stored

Object-Level Attributes		
Attribute and Default	Legal Values	Definition
matrix float matrix[16]		This attribute is the 4x4 transformation matrix which should be applied to the location coordinates of the object. The attribute <code>has_matrix</code> must also be set to <code>TRUE</code> to indicate that there is a matrix present. Persistence: stored
max float max[3]		This attribute is the maximum corner of the bounding box which defines extent of the geometry contained in this object. The attribute <code>min</code> must also be set along with this attribute to fully define the bounding box. The attribute <code>has_bounding_box</code> must also be set to <code>TRUE</code> to indicate that there is a bounding box present. Persistence: stored
min float min[3]		This attribute is the minimum corner of the bounding box which defines extent of the geometry contained in this object. The attribute <code>max</code> must also be set along with this attribute to fully define the bounding box. The attribute <code>has_bounding_box</code> must also be set to <code>TRUE</code> to indicate that there is a bounding box present. Persistence: stored
modeling_space float modeling_space <code>KWORLD_SPACE</code>	<code>KWORLD_SPACE</code> <code>KNDC_SPACE</code>	The modeling space attribute defines the coordinate space in which the location data is defined. A value of <code>KNDC_SPACE</code> implies that all location data should be in normalized device coordinates and will range between 0.0 and 1.0. Persistence: stored
opacity float opacity 1.0	0.0 < 1.0	The opacity attribute is a real number between 0.0 and 1.0 where 0.0 implies that the object is completely transparent. It should only be interpreted if the <code>has_alpha</code> attribute is set to <code>TRUE</code> . Persistence: stored

Object-Level Attributes		
Attribute and Default	Legal Values	Definition
<pre>specular_color float specular_color[3] [.25 .25 .25]</pre>	0.0 < 1.0	<p>An object's specular reflectance is combined and like diffuse reflection, is brighter as the incident angle nears perpendicular. However, unlike diffuse reflectance, specular reflectance is reflected away from the object in a particular direction (the reflected direction is the mirror angle of the angle between the surface normal and this incident direction). These values indicate the amount of reflected specular light in each of the R, G, and B components. The values may range from 0.0 and 1.0 where 0.0 implies no specular reflection.</p> <p>Persistence: stored</p>
<pre>specular_exponent float specular_exponent 10.0</pre>	0.0 < 200.0	<p>Controls the sharpness of specular highlights</p> <p>Persistence: stored</p>
<pre>texture_coord_dim int texture_coord_dim 2</pre>	> 1	<p>This attribute specifies the dimensionality of the texture coordinate vertices. A dimension of 2 would imply a 2D space with locations specified in u and v. A dimension size of 3 would imply a 3D space with locations specified in u, v, and w.</p> <p>Persistence: stored</p>
<pre>visible int visible TRUE</pre>	TRUE FALSE	<p>The boolean attribute is a flag which specifies whether the object is visible or not.</p> <p>Persistence: stored</p>

A.4. Geometry Primitives and Associated Attributes

The data associated with each primitive consists of multiple data components. For example, a KGEOM_SPHERES primitive consists of location data, color data, and radii data. These data components are assigned to the primitive structure.

The following table presents each primitive in turn, first listing the data pointers, then specifying the primitive's attributes. Note that the double lines in the table delineate these two parts of the primitive specification.

Table 12: Geometry Primitives and Attributes		
Primitive Name	Component or Attribute	Description
KGEOM_POLYLINE_DISCONNECTED KGEOM_POLYLINE_CONNECTED		A disjoint polyline KGEOM_POLYLINE_DISCONNECTED with N vertices results in N/2 lines. A connected polyline KGEOM_POLYLINE_CONNECTED with N vertices results in N-1 lines.
	float * locs	An array of vertex locations. The dimension of the locations is set by the <code>location_dim</code> attribute. The number of locations is specified by the <code>nverts</code> attribute.
	float * cols	An optional array of colors. The color data will contain either RGB or RGBA vectors depending on how the <code>has_alpha</code> attribute is set. The number of colors will be affected by the <code>layout</code> attribute.
	int nverts	This value specifies the number of vertices. Default: unspecified
	int linetype	This attribute specifies the line type. Default: 1
	int linewidth	This attribute specifies the line width. Default: 1
	KGEOM_TRIANGLES_DISJOINT KGEOM_TRIANGLES_CONNECTED	
float * locs		An array of vertex locations. The dimension of the locations is set by the <code>location_dim</code> attribute. The number of locations is specified by the <code>nverts</code> attribute.
float * cols		An optional array of colors. The color data will contain either RGB or RGBA vectors depending on how the <code>has_alpha</code> attribute is set. The number of colors will be affected by the <code>layout</code> attribute.
float * norms		An array of normals. The number of normals will be affected by the <code>layout</code> attribute. The dimensionality of the normals is set by the <code>location_dim</code> attribute.
float * tcoords		An array of texture coordinates. The dimensionality of the texture coordinates is specified by the <code>texture_coord_dim</code> attribute.

Table 12: Geometry Primitives and Attributes

Primitive Name	Component or Attribute	Description
	int nverts	This attribute specifies the number of vertices in the connected or disjoint polytriangles. Default: unspecified
	char * texture	This attribute will provide a file name from which the KGEOM_TEXTURE2D primitive associated with this primitive can be obtained. Default: NULL
KGEOM_SPHERES		A list of spheres, where the number of spheres is equal to the value specified by the nverts attribute.
	float * locs	An array of vertex locations. The dimension of the locations is set by the location_dim attribute. The number of locations is specified by the nverts attribute.
	float * cols	An optional array of colors. The color data will contain either RGB or RGBA vectors depending on how the has_alpha attribute is set.
	float * radii	A 1D array of sphere radii.
	int nverts	This attribute specifies the number of sphere centers. Default: unspecified
KGEOM_CYLINDERS		A list of cylinders, where the number of cylinders is equal to nverts / 2.
	float * locs	An array of vertex locations. The dimension of the locations is set by the location_dim attribute. The number of locations is specified by the nverts attribute.
	float * rad1 float * rad2	Arrays of cylinder top and bottom radii.
	float * cols	An optional array of colors. The color data will contain either RGB or RGBA vectors depending on how the has_alpha attribute is set. The number of colors will be affected by the layout attribute.
	int nverts	This value is the number of cylinders times two. Default: unspecified
	KGEOM_DIRECTED_POINTS	

Table 12: Geometry Primitives and Attributes

Primitive Name	Component or Attribute	Description
	float * locs	An array of vertex locations. The dimension of the locations is set by the <code>location_dim</code> attribute. The number of locations is specified by the <code>nverts</code> attribute.
	float * cols	An optional array of colors. The color data will contain either RGB or RGBA vectors depending on how the <code>has_alpha</code> attribute is set.
	float * norms	An array of normals. The dimensionality of the normals is set by the <code>location_dim</code> attribute.
	int nverts	This attribute specifies the number of directed points. Default: unspecified
KGEOM_TEXT		A list of null terminated strings.
	kstring * strings	An array of strings.
	kstring * fonts	An array of font names, one for each string.
	float * locs float * offsets float * ups float * flows	There is an array each for text location, offsets, ups, and flows. The dimensionality of the locations is set by the <code>location_dim</code> attribute. Each string has a location vector, an offset vector, an up vector, and a flow vector which indicates what direction to draw the text in.
	float * cols float * bgcols	There are optional arrays of text foreground and background colors, one foreground and background color vector for each string. The color data will contain either RGB or RGBA vectors depending on how the <code>has_alpha</code> attribute is set.
	int nverts	This attribute specifies the number of strings. Default: unspecified
KGEOM_MARKERS		A list of markers.
	float * locs	An array of vertex locations. The dimension of the locations is set by the <code>location_dim</code> attribute. The number of locations is specified by the <code>nverts</code> attribute.
	float * cols float * bgcols	There are optional arrays of marker foreground and background colors, one foreground and background color vector for each marker. The color data will contain either RGB or RGBA vectors depending on how the <code>has_alpha</code> attribute is set.

Table 12: Geometry Primitives and Attributes

Primitive Name	Component or Attribute	Description
	int nverts	This attribute specifies the number of markers. Default: unspecified
	int makertype	This attribute can have the value of KMARKER_SQUARE, KMARKER_TRIANGLE, KMARKER_CROSS, KMARKER_BOW_TIE, KMARKER_ARC, KMARKER_DIAMOND, KMARKER_CIRCLE, KMARKER_V, KMARKER_HEXAGON, KMARKER_X, KMARKER_DOT, KMARKER_CARET, KMARKER_POINT, KMARKER_DAGGER, or KMARKER_BOX. Default: KMARKER_SQUARE
KGEOM_POLYGON		A single polygon, where the value of the nverts attribute specifies the number of vertices in the polygon, which is one more than the number of line segments in the polygon.
	float * locs	An array of vertex locations. The dimension of the locations is set by the location_dim attribute. The number of locations is specified by the nverts attribute.
	float * cols float * bgcol	There is an optional array of polygon foreground colors. The number of foreground colors will either be equal to nverts There is also a single optional background color. The color data will contain either RGB or RGBA vectors depending on how the has_alpha attribute is set.
	int nverts	This attribute specifies the number of vertices in the polygon. Default: unspecified
	int linetype	This attribute specifies the line type. Default: 1
	int linewidth	This attribute specifies the line width. Default: 1
KGEOM_RECTANGLES		A list of rectangles, where the value of the nverts attribute specifies the number of rectangles divided by two.
	float * locs	Each rectangle has a minimum corner vertex and a maximum corner vertex. Each vertex has a location vector. The dimension of the locations is set by the location_dim attribute. The number of locations is specified by the nverts attribute.

Table 12: Geometry Primitives and Attributes

Primitive Name	Component or Attribute	Description
	float * cols float * bgcols	Each rectangle has an optional line color vector and a fill color vector. The color data will contain either RGB or RGBA vectors depending on how the <code>has_alpha</code> attribute is set.
	int nverts	This attribute specifies the number of vertices in the rectangle list. The number of rectangles is <code>nverts / 2</code> . Default: unspecified
	int linetype	This attribute specifies the line type. Default: 1
	int linewidth	This attribute specifies the line width. Default: 1
KGEOM_ELLIPSES		A list of ellipses, where the value of the <code>nverts</code> attribute specifies the number of ellipses.
	float * locs	Each ellipse has a center vertex location vector. The dimension of the locations is set by the <code>location_dim</code> attribute. The number of locations is specified by the <code>nverts</code> attribute.
	float * rad1 float * rad2	Each ellipse has a major and minor axis. The number of axis vectors is given by the value of the <code>nverts</code> attribute.
	float * cols float * bgcols	Each ellipse has an optional line color vector and a fill color vector. The color data will contain either RGB or RGBA vectors depending on how the <code>has_alpha</code> attribute is set.
	int nverts	This attribute specifies the number of vertices in the list of ellipses. Default: unspecified
	int linetype	This attribute specifies the line type. Default: 1
	int linewidth	This attribute specifies the line width. Default: 1
KGEOM_QUADMESH		A 2D mesh of data.
	float * locs	Quadmshes consist of <code>width</code> by <code>height</code> locations. The dimension of the locations is set by the <code>location_dim</code> attribute. The number of locations is specified by the <code>nverts</code> attribute.
	float * cols	An optional array of colors. The color data will contain either RGB or RGBA vectors depending on how the <code>has_alpha</code> attribute is set. The number of colors will be affected by the <code>layout</code> attribute.

Table 12: Geometry Primitives and Attributes

Primitive Name	Component or Attribute	Description
	float * norms	An array of normals. The number of normals will be affected by the <code>layout</code> attribute. The dimensionality of the normals is set by the <code>location_dim</code> attribute.
	float * tcoords	An array of texture coordinates. The dimensionality of the texture coordinates is specified by the <code>texture_coord_dim</code> attribute.
	int width height	These attributes specify the number of points available in the quadmesh. Default: unspecified
	char * texture	This attribute will provide a file name from which the <code>KGEOM_TEXTURE2D</code> primitive associated with this primitive can be obtained. Default: NULL
KGEOM_OCTMESH		A 3D mesh of data.
	float * locs	Octmeshes consist of width by height by depth locations. The dimension of the locations is set by the <code>location_dim</code> attribute. The number of locations is specified by the <code>nverts</code> attribute.
	kubyte * cols	An optional array of colors. The color data will contain either RGB or RGBA vectors depending on how the <code>has_alpha</code> attribute is set. The number of colors will be affected by the <code>layout</code> attribute.
	float * norms	An array of normals. The number of normals will be affected by the <code>layout</code> attribute. The dimensionality of the normals is set by the <code>location_dim</code> attribute.
	float * tcoords	An array of texture coordinates. The dimensionality of the texture coordinates is specified by the <code>texture_coord_dim</code> attribute.
	int width height depth	These attributes specify the number of points available in the quadmesh. Default: unspecified
	char * texture	This attribute will provide a file name from which the <code>KGEOM_TEXTURE2D</code> primitive associated with this primitive can be obtained. Default: NULL
KGEOM_TEXTURE2D		A 2D texture of colors.

Table 12: Geometry Primitives and Attributes

Primitive Name	Component or Attribute	Description
	kubyte * cols	An array of colors. The color data will contain either RGB or RGBA vectors depending on how the <code>has_alpha</code> attribute is set.
	int width height	These attributes specify the number of points available in the 2D texture. Default: unspecified
KGEOM_TEXTURE3D		A 3D texture of colors.
	kubyte * cols	An array of colors. The color data will contain either RGB or RGBA vectors depending on how the <code>has_alpha</code> attribute is set.
	int width height depth	These attributes specify the number of points available in the 3D texture. Default: unspecified
KGEOM_TEXTURE2D		A 2D texture of colors.
	kubyte * cols	An array of colors. The color data will contain either RGB or RGBA vectors depending on how the <code>has_alpha</code> attribute is set.
	int width height	These attributes specify the number of points available in the 2D texture. Default: unspecified
KGEOM_VIEW_XFORM		A viewing transformation.
	float matrix[16]	A transformation matrix to apply to the current viewing position. Default: identity matrix
KGEOM_OBJECT_XFORM		A transformation to be applied to a particular object.
	kstring name	The name of the object to apply the transformation to. Default: NULL
	float matrix[16]	The transformation matrix to apply to the object's locations. Default: identity matrix
	float translate[3]	An additional translation to apply to the object. Default: [0.0 0.0 0.0]
	int relative	Indicates if the transformation is relative to the current position or specifies an absolute transformation. Default: KRELATIVE
KGEOM_CAMERA_ORIENTATION		A camera specification to use when rendering.
	float eye[3]	The current eye position of the camera. Default: [0.0 0.0 0.0]

Table 12: Geometry Primitives and Attributes

Primitive Name	Component or Attribute	Description
	float at [3]	The current position to point the camera at. Default: [0.0 0.0 1.0]
	float up [3]	The current up direction Default: [0.0 1.0 0.0]
	float field_of_view	Specifies the field of view of the camera in degrees. Default: 90.0
	float hither	Specifies the near clip plane. Default: -1.0
	float yon	Specifies the far clip plane. Default: 1.0
	int projection	Specifies if the rendering should be done with a parallel or perspective projection. Default: KPROJECTION_PERSPECTIVE
	int is_stereo	Specifies if stereo rendering should be done. Default: FALSE
	float eye_distance	Specifies the distance between the two stereo cameras. Default: 0.0
	float focal_length	Specifies the distance to the intersection between the viewing directions of the the two stereo cameras. Default: 1.0
KGEOM_VR_EVENT		An event from a virtual reality device.
	int have_button_events	This is TRUE if any button events are present. Default: FALSE
	int have_rotations	This is TRUE if any rotation events are present Default: FALSE
	int have_translations	This is TRUE if any translation events are present Default: FALSE
	float rotate [3]	This specifies an XYZ rotation. Default: [0.0 0.0 0.0]
	float translate [3]	This specifies an XYZ translation. Default: [0.0 0.0 0.0]
	int buttons [9]	This specifies the state of 9 buttons on a VR device. A value of 1 implies that the button is pressed. Default: [0 0 0 0 0 0 0 0 0]
	int clock	A clock value from the VR device. Default: 0
KGEOM_OBJECT		Another geometry object

Table 12: Geometry Primitives and Attributes		
Primitive Name	Component or Attribute	Description
	<code>kgeom_object *</code> object	This primitive will contain a pointer to another geometry object. In general, this primitive should not be used. It will be hidden in the use of the <code>kgeom_add_primitive()</code> and <code>kgeom_get_primitive()</code> functions when they are used with another geometry object. Default: NULL

A.5. Geometry Service Functions

A.5.1. Object Functions

Geometry data and attributes are stored and retrieved from a geometry object. The following functions allow you to create a new geometry object, write and read a geometry object to and from a file or transport, copy a geometry object, or free a geometry object. Geometry objects are represented by pointers to the data structure `kgeom_object`.

- `kgeom_new_object()` - construct a new geometry object
- `kgeom_write_object()` - write a geometry object
- `kgeom_read_object()` - read a geometry object
- `kgeom_copy_object()` - copy a geometry object
- `kgeom_blast_object()` - free a geometry object

A.5.2. `kgeom_new_object()` — *construct a new geometry object*

Synopsis

```
kgeom_object *
kgeom_new_object(void)
```

Returns

a pointer to the constructed object on success, NULL on failure

Description

This function will construct a new geometry object structure and initialize all of its fields with their default values.

Once constructed, the fields of this geometry object can be assigned specific values. For a complete

explanation of these fields, see the *kgeom_object* man page.

Geometry Primitives and Primitive Lists

A geometry primitive contains geometric data which describes a shape in space. Lines, triangles, and spheres, are all considered to be geometric primitives. For a complete description of all the geometry primitives, see the *kgeom_primitive* man page.

Each geometry object contains a list of such geometry primitives. New primitives can be created using the function `kgeom_new_primitive()`. Once a new primitive has been created and its data assigned, it can be added to the primitive list of the object using the function `kgeom_add_primitive()`. A corresponding function `kgeom_remove_primitive()` allows you to remove a primitive from the primitive list of an object.

The number of primitives which have been added to the primitive list of an object is returned by the function `kgeom_number_primitives()`. A primitive at a specific position in the primitive list can be retrieved using the function `kgeom_get_primitive()`.

The primitive functions should be the only means used for accessing the primitives on a primitive list. Even though the primitive list of the geometry object could be accessed directly, it should be considered to be private. By using the primitive list function calls, you can be certain to maintain the integrity of the primitive list.

Reading and Writing Geometry Objects

A geometry object can be written out to a file, or other transport using the function `kgeom_write_object()`. This function will write out all the object and primitive information to a specified file.

Geometry that has been stored in a file can be read back with the function `kgeom_read_object()`. This function will construct a new object into which it will read the geometry; it is not necessary to pre-construct an object with the `kgeom_new_object()` function prior to reading.

Other Geometry Object Functions

A geometry object can be copied using the function `kgeom_copy_object()`.

A geometry object can be destroyed with a call to the function `kgeom_geom_blast_object()`. This will destroy the object and all of its primitives, freeing all associated memory. Be careful not to access any of the geometry object's data after blasting it.

Examples

This simple code illustrates how a new geometry object would be constructed :

```
kgeom_object *obj;
```

```
obj = kgeom_new_object();
```

A.5.3. `kgeom_write_object()` — *write a geometry object*

Synopsis

```
int  
kgeom_write_object(kgeom_object *object, char *filename)
```

Input Arguments

```
object  
    geometry object to write  
filename  
    filename to write geometry object to.
```

Returns

TRUE on success, FALSE otherwise

Description

This function will write the given geometry object to a specified file. The object can later be read from that file using the `kgeom_read_object()` function.

This function will write everything contained in the geometry object to the file. All the object information will be written followed by the information and data contained in each primitive in the object's primitive list.

After the object has been written, you may free it using the `kgeom_blast_object()` function.

Examples

The following code would be used to write a geometry object out to the file "data.kgm":

```
kgeom_object *geom;  
  
// presumably some primitives would be added here  
  
kgeom_write_object(geom, "data.kgm");
```

A.5.4. kgeom_read_object() — *read a geometry object*

Synopsis

```
kgeom_object *  
kgeom_read_object(char *filename)
```

Input Arguments

```
filename  
filename to read geometry from
```

Returns

pointer to a new geometry object containing the geometry data contained in the file, NULL otherwise

Description

This function will read an entire geometry object object from the specified file.

If the file does not contain a valid geometry object, then it will be closed and nothing will be returned. If it does contain a valid header, then a new object containing the geometry data from the file will be returned.

Examples

The following code would be used to read in a geometry object from the file "data.kgm":

```
kgeom_object *geom;  
  
geom = kgeom_read_object("data.kgm");
```

A.5.5. `kgeom_copy_object()` — *copy a geometry object*

Synopsis

```
kgeom_object *  
kgeom_copy_object(kgeom_object *object, kgeom_object *new_object)
```

Input Arguments

`object`
geometry object to copy

Output Arguments

`new_object`
a pointer to geometry object that will serve as a destination for the copy. If NULL, then a new destination object will be allocated.

Returns

copy of object

Description

This function will copy the given geometry object, creating a duplicate of the object as well as of all the object's primitives. All the data contained in each of the object's primitives will be copied as well.

If desired, a preallocated object can be provided as the destination for the copy. Any data or primitives which may have existed in the destination prior to calling this routine will be destroyed.

If no destination object is provided, one will be allocated and returned.

A.5.6. `kgeom_blast_object()` — *free a geometry object*

Synopsis

```
void  
kgeom_blast_object(kgeom_object *object)
```

Input Arguments

`object`
geometry object to destroy

Description

This function will destroy the given geometry object, first freeing all the primitives on the object's primitive list, and then freeing the object itself.

Note that geometry services will free *all* the primitives and data pointers within the geometry object. Since geometry services will try to free all memory pointed to by the geometry object, be careful not to place any static primitives or point to static arrays of data from within the geometry object. If geometry services tries to free a statically allocated pieces of memory, it will result in a fatal error.

If you wish to keep any component of the geometry object in memory, simply remove it from the geometry object prior to calling this function. For instance, to keep a primitive from an object in memory after the object has been destroyed, that primitive could first be removed with the `kgeom_remove_primitive()` function. Similarly, to keep a particular array of data from a primitive in memory after the associated primitive has been destroyed, the pointer to that array could be assigned to NULL. This routine will only free the primitives and data pointers which are seen within the object.

A.5.7. Primitive Functions

Geometry data is stored specifically in geometry primitive structures. The following functions are available for creating, copying, and freeing geometry primitives.

- `kgeom_new_primitive()` - construct a new geometry primitive
- `kgeom_copy_primitive()` - copy a geometry primitive
- `kgeom_blast_primitive()` - destroy a geometry primitive

A.5.8. `kgeom_new_primitive()` — *construct a new geometry primitive*

Synopsis

```
kgeom_primitive *  
kgeom_new_primitive(int type)
```

Input Arguments

```
type  
    type of primitive to construct
```

Returns

a pointer to the constructed primitive on success, NULL on failure

Description

This function will construct a new geometry primitive structure of the given type and initialize all default values.

A geometry primitive contains geometric data which describes a shape in space. Lines, triangles, and spheres, are all considered to be geometric primitives. Each "type" of primitive is actually represented by a distinct structure; these structures have been unioned into a common `kgeom_primitive` structure.

The type of primitive is declared on instantiation, as in this example :

```
kgeom_primitive *prim;  
  
prim = kgeom_new_primitive(KGEOM_POLYLINE_CONNECTED);
```

The type of primitive is tracked in a type field which is common to all the primitive structures. It can always be accessed as `prim->type` from any primitive structure.

For a complete description of all the types of geometry primitives, see the *kgeom_primitive* man page, or consult the Data Services Manual : Programming Services Volume II

Note that a geometry primitive is not limited to a single instance of a shape. For example, a single spheres primitive can contain many spheres and a single polyline primitive can contain many lines.

The data within a primitive consists of several components; the predominate component which actually defines the size and position of the geometry is the location data. The location data consists of a list of vertices. The number of these vertices is a specified by a field within the primitive and is a defining characteristic for most geometry primitives.

Additional components which may be present in a geometry primitive are color, normals, or texture coordinates. All components except for the location component are optional in any given geometry primitive.

Most the fields of a geometry primitive are pointers to allocated arrays which contain these components. The size of these arrays is implied from the number of vertices present in the geometry primitive, and the presentation characteristics (`layout`, `location_dim`, `texture_coord_dim`, and `has_alpha`) set

in the associated geometry object. Most data within a geometry primitive will be floating point.

Geometric data is always stored "elements first", with the vectors forming the leading dimension of the array. For example, 3D location vertices will be stored in the order XYZXYZXYZ...

The number of location vertices is by definition the number of vertices in the primitive. The size of each vertex will be determined by the location_dim field of the associated object.

The number of color and normal vectors will be a function of the number of vertices, but will vary depending on the primitive and the layout field of associated geometry object. For example, a disjoint polyline primitive will have 1 color per vertex for a KPER_VERTEX layout, but will have 1 color per line for a KPER_LINE primitive.

The size of each color vector will either be 3 (for RGB), of 4 (for RGBA) if the has_alpha field of the associated object is TRUE. The size of each normal vector will match the size of location_dim field in the associated object.

If texture coordinate vectors are present, the number of texture coordinates must match the number of vertices in the primitive. The size of each texture coordinate will be determined by the texture_coord_dim field of the associated object.

If the modeling_space field of the associated object is set to KNDC_SPACE, the location data will be interpreted as normalized device coordinates and thus should range between 0.0 and 1.0.

Color data is represented by RGB vectors with three floats determining the red, green, and blue intensity. These numbers should range from 0.0 to 1.0, with 1.0 implying maximum intensity. If an alpha component is present it will determine the transparency of the primitive. It also should range from 0.0 to 1.0, with 0.0 implying that the primitive is totally transparent, and 1.0 implying that the primitive is totally opaque.

Since the fields of a geometry primitive vary from type to type, we must specify the type of primitive we are working with in order to access the components of the primitive structure. For example, if we had a spheres primitive, we would have to do the following to access the specific sphere fields :

```
kgeom_primitive *prim;  
  
prim = kgeom_new_primitive(KGEOM_SPHERES);  
  
// specify that we want 45 spheres  
  
prim->spheres.nverts = 45;
```

This can alternatively be done by casting the structure to its specific primitive type as follows :

```
kgeom_spheres *s;  
  
s = (kgeom_spheres *) kgeom_new_primitive(KGEOM_SPHERES);
```

```
// specify that we want 45 spheres  
  
s->nverts = 45;
```

Primitives can be cast to any type at any time, but you should be sure that you cast back to the general `kgeom_primitive *` type before passing a specific primitive into any functions which take a primitive argument.

A.5.9. `kgeom_copy_primitive()` — *copy a geometry primitive*

Synopsis

```
kgeom_primitive *  
kgeom_copy_primitive(  
    kgeom_object *object,  
    kgeom_primitive *primitive,  
    kgeom_primitive *new_primitive)
```

Input Arguments

`object`
object to which original primitive belongs
`primitive`
primitive to copy

Output Arguments

`new_primitive`
a pointer to geometry primitive that will serve as a destination for the copy. If NULL, then a new destination primitive will be allocated.

Returns

copy of primitive

Description

This routine will copy a given primitive and all associated data into another primitive structure. If another primitive structure is not provided for the destination of the copy, a new one will be constructed.

The object presentation is used to determine how much data is present in the primitive, so make certain the `has_alpha`, `layout`, `location_dim`, and other fields correctly reflect the data in the primitive. The copied primitive will not be added to the object, so you are free to destroy it or add it to another object.

A.5.10. `kgeom_blast_primitive()` — *destroy a geometry primitive*

Synopsis

```
int  
kgeom_blast_primitive(kgeom_primitive *primitive)
```

Input Arguments

```
primitive  
primitive to free
```

Returns

TRUE on success, FALSE otherwise

Description

This routine will free a given primitive and all associated data.

Note that geometry services will free *all* the data pointers within the geometry primitive. Because of this, be careful not to point to static arrays of data within the geometry primitive. If geometry services tries to free a statically allocated array, it will result in a fatal error.

A.5.11. Primitive List Functions

Geometry data is stored and retrieved from a *primitive list* contained within a geometry object. The following functions allow you to add and remove primitives from a primitive list, as well as determine the number of primitives contained therein.

- `kgeom_add_primitive()` - add a primitive to a geometry object
- `kgeom_get_primitive()` - get a primitive from a geometry object
- `kgeom_number_primitives()` - count the number of primitives in the given object

A.5.12. `kgeom_add_primitive()` — *add a primitive to a geometry object*

Synopsis

```
int  
kgeom_add_primitive(  
    kgeom_object *object,  
    kgeom_primitive *primitive)
```

Input Arguments

object
object to add primitive to
primitive
primitive to add

Returns

TRUE on success, FALSE on failure

Description

This function will add the given primitive to the end of the primitive list of the given object.

Geometry Primitives and Primitive Lists

A geometry primitive contains geometric data which describes a shape in space. Lines, triangles, and spheres, are all considered to be geometric primitives. For a complete description of all the geometry primitives, see the *kgeom_primitive* man page.

Each geometry object contains a list of such geometry primitives. New primitives can be created using the function `kgeom_new_primitive()`. The function `kgeom_remove_primitive()` allows you to remove a primitive from the primitive list of an object after it has been added. At any time, the number of primitives which have been added to the primitive list of an object can be determined by the function `kgeom_number_primitives()`. A primitive at any given position can be retrieved using the function `kgeom_get_primitive()`.

The primitive functions should be the only means used for accessing the primitives on a primitive list. Even though the primitive list of the geometry object could be accessed directly, it should be considered to be private. By using the primitive list function calls, you can be certain to maintain the integrity of the primitive list.

A.5.13. `kgeom_get_primitive()` — *get a primitive from a geometry object*

Synopsis

```
kgeom_primitive *  
kgeom_get_primitive(  
    kgeom_object *object,  
    int           position)
```

Input Arguments

object
object to retrieve primitive from
position
position in list from which the primitive should be retrieved

Returns

TRUE on success, FALSE on failure

Description

This routine will get the primitive from the specified position in the primitive list. If the position is outside the bounds of the primitive list then a NULL pointer will be returned.

Geometry Primitives and Primitive Lists

A geometry primitive contains geometric data which describes a shape in space. Lines, triangles, and spheres, are all considered to be geometric primitives. For a complete description of all the geometry primitives, see the *kgeom_primitive* man page.

Each geometry object contains a list of such geometry primitives. New primitives can be created using the function `kgeom_new_primitive()`. Once a new primitive has been created and its data assigned, it can be added to the primitive list of the object using the function `kgeom_add_primitive()`. A corresponding function `kgeom_remove_primitive()` allows you to remove a primitive from the primitive list of an object. The number of primitives which have been added to the primitive list of an object is returned by the function `kgeom_number_primitives()`.

The primitive functions should be the only means used for accessing the primitives on a primitive list. Even though the primitive list of the geometry object could be accessed directly, it should be considered to be private. By using the primitive list function calls, you can be certain to maintain the integrity of the primitive list.

A.5.14. `kgeom_number_primitives()` — *count the number of primitives in the given object*

Synopsis

```
int  
kgeom_number_primitives(kgeom_object *object)
```

Input Arguments

```
object  
object to count primitives in
```

Returns

the number of primitives contained in the object

Description

This function will return the number of primitives contained in the primitive list of the given object.

Geometry Primitives and Primitive Lists

A geometry primitive contains geometric data which describes a shape in space. Lines, triangles, and spheres, are all considered to be geometric primitives. For a complete description of all the geometry primitives, see the *kgeom_primitive* man page.

Each geometry object contains a list of such geometry primitives. New primitives can be created using the function `kgeom_new_primitive()`. The function `kgeom_remove_primitive()` allows you to remove a primitive from the primitive list of an object after it has been added. A primitive at any given position can be retrieved using the function `kgeom_get_primitive()`.

The primitive functions should be the only means used for accessing the primitives on a primitive list. Even though the primitive list of the geometry object could be accessed directly, it should be considered to be private. By using the primitive list function calls, you can be certain to maintain the integrity of the primitive list.

A.5.15. Specialized Reading and Writing Functions

The following functions are used for doing incremental reads and writes of a data object. In general, these functions should *only* be used if you wish to keep a single geometry primitive in memory at one time. The `kgeom_write_object` and `kgeom_read_object` will call these functions internally.

- `kgeom_start_writing_object()` - write the first part of a geometry object
- `kgeom_write_primitive()` - write a geometry primitive
- `kgeom_finish_writing_object()` - write the last part of a geometry object
- `kgeom_done_writing()` - close associated file after writing

- `kgeom_start_reading_object()` - read the first part of a geometry object
- `kgeom_read_primitive()` - read a geometry primitive
- `kgeom_finish_reading_object()` - read the last part of a geometry object
- `kgeom_done_reading()` - close associated file after reading

A.5.16. <code>kgeom_start_writing_object()</code> — <i>write the first part of a geometry object</i>

Synopsis

```
int  
kgeom_start_writing_object(kgeom_object *object)
```

Input Arguments

```
object  
  geometry object to write
```

Returns

TRUE on success, FALSE otherwise

Description

This routine will write the first part of a geometry object to the open fid. It should be followed by a write of all the primitives, and then by a finish of the object.

In general, the `kgeom_write_object()` function should be used instead of this function. This function has been made public to provide complete flexibility for writing a geometry object. It should only be used when you wish to write a geometry object primitive-by-primitive. More information on this is available in the `kgeom_write_primitive()` man page.

The first part of a geometry object contains information which will be needed to later read the geometry primitive data. Specifically, the `layout`, `location_dim`, `texture_coord_dim`, and `has_alpha` fields are written as the first part of the object.

All other information contained within the geometry object is written after the primitives by the `kgeom_finish_writing_object()` call.

The fid internal to the geometry object specifies the file to which the the object will be written. In general, this fid is set when the geometry header is written by the `kgeom_write_header()` function.

A.5.17. `kgeom_write_primitive()` — *write a geometry primitive*

Synopsis

```
int
kgeom_write_primitive(
    kgeom_object *object,
    kgeom_primitive *primitive)
```

Input Arguments

`object`
object to which primitive belongs

`primitive`
primitive to write

Returns

TRUE on success, FALSE otherwise

Description

This routine will write a given primitive and all associated data to the file indicated by the provided object.

In general, the `kgeom_write_object()` function should be used instead of this function. This function has been made public to provide complete flexibility for writing a geometry object. It should only be used when you wish to write a geometry object primitive-by-primitive.

If you wish to use this function to write out a primitive at a time, you should follow this sequence to write out the the object:

```
kgeom_write_header(object, "filename");  
  
kgeom_start_writing_object(object);
```

At this point, you can start writing out individual primitives using the `kgeom_write_primitive()` function. The file to be written to is determined from the open fid inside the object.

When you are done writing primitives, finish with the following calls :

```
kgeom_finish_writing_object(object);  
  
kgeom_done_writing(object);
```

The object presentation is used to determine how much data is present in the primitive, so make sure the `has_alpha`, `layout`, `location_dim`, and other fields are set correctly. These should be valid before the call to `kgeom_start_writing_object()` is made. The primitive does not have to be present on the object's primitive list to be written.

A.5.18. `kgeom_finish_writing_object()` — *write the last part of a geometry object*

Synopsis

```
int  
kgeom_finish_writing_object(kgeom_object *object)
```

Input Arguments

```
object  
geometry object to write
```

Returns

TRUE on success, FALSE otherwise

Description

This routine will terminate the primitive list by writing a -1 to an open fid. It will then write out all the

object specific information.

In general, the `kgeom_write_object()` function should be used instead of this function. This function has been made public to provide complete flexibility for writing a geometry object. It should only be used when you wish to write a geometry object primitive-by-primitive. More information on this is available in the `kgeom_write_primitive()` man page.

The first part of a geometry object should have already been written by a call to `kgeom_start_writing_object()`. That call would have written any information which would be needed to later read the geometry primitive data. Specifically, the `layout`, `location_dim`, `texture_coord_dim`, and `has_alpha` fields of the object would already have been written out as the first part of the object.

All other information contained within the geometry object is written after the primitives by this call.

The fid internal to the geometry object specifies the file to which the the object will be written. In general, this fid is set when the geometry header is written by the `kgeom_write_header()` function.

A.5.19. `kgeom_done_writing()` — *close associated file after writing*

Synopsis

```
int kgeom_done_writing(kgeom_object *object)
```

Input Arguments

`object`
geometry object to write filename - filename to write header to

Returns

TRUE on success, FALSE otherwise

Description

This function will close the file associated with the geometry object after a write has been completed.

In general, the `kgeom_write_object()` function should be used instead of this function. This function has been made public to provide complete flexibility for writing a geometry object. It should only be used when you wish to write a geometry object primitive-by-primitive. More information on this is available in the `kgeom_write_primitive()` man page.

The fid internal to the geometry object specifies the file which will be closed. In general, this fid is set when the geometry header was written by the `kgeom_write_header()` function.

A.5.20. `kgeom_start_reading_object()` — *read the first part of a geometry object*

Synopsis

```
int  
kgeom_start_reading_object(kgeom_object *object)
```

Input Arguments

```
object  
  geometry object to read
```

Returns

TRUE on success, FALSE otherwise

Description

This routine will read the first part of a geometry object from an open fid. It should be followed by a read of all the primitives, and then by a finish of the object.

In general, the `kgeom_read_object()` function should be used instead of this function. This function has been made public to provide complete flexibility for reading a geometry object. It should only be used when you wish to read a geometry object primitive-by-primitive. More information on this is available in the `kgeom_read_primitive()` man page.

The first part of a geometry object contains information which is needed to read the geometry primitive data. Specifically, the `layout`, `location_dim`, `texture_coord_dim`, and `has_alpha` fields are read in as the first part of the object.

All other information contained within the geometry object is read after the primitives by the `kgeom_finish_reading_object()` call.

The fid internal to the geometry object specifies the file from which the the object will be read. In general, this fid is set when the geometry header is read by the `kgeom_read_header()` function.

A.5.21. `kgeom_read_primitive()` — *read a geometry primitive*

Synopsis

```
kgeom_primitive *  
kgeom_read_primitive(kgeom_object *object)
```

Input Arguments

```
object  
  object to which primitive will belong
```


Returns

new primitive with primitive data from file

Description

This routine will read a given primitive and all associated data from the file indicated by the provided object. A new primitive will be constructed for this.

In general, the `kgeom_write_object()` function should be used instead of this function. This function has been made public to provide complete flexibility for reading a geometry object. It should only be used when you wish to read a geometry object primitive-by-primitive.

If you wish to use this function to read in a primitive at a time, you should follow this sequence to reading the object:

```
object = kgeom_read_header("filename");
```

If this has returned a valid object, then the object itself can be read. The next call will read in all the specific information

```
kgeom_start_reading_object(object);
```

At this point, you can start reading in individual primitives using the `kgeom_read_primitive()` function. The file to be read from is determined from the open fid inside the object. This function will return NULL if there are no more primitives to read.

After all primitives have been read, finish with the following calls :

```
kgeom_finish_reading_object(object);
```

```
kgeom_done_reading(object);
```

The object presentation is used to determine how much data is present in the primitive. The call to `kgeom_start_reading_object()` should have initialized these fields appropriately. The primitive will not be added to the object, so you are free to destroy it or add it to a different object if you choose.

A.5.22. `kgeom_finish_reading_object()` — *read the last part of a geometry object*

Synopsis

```
int  
kgeom_finish_reading_object(kgeom_object *object)
```

Input Arguments

object
geometry object to read

Returns

TRUE on success, FALSE otherwise

Description

This routine will read in all the object specific information.

In general, the `kgeom_read_object()` function should be used instead of this function. This function has been made public to provide complete flexibility for reading a geometry object. It should only be used when you wish to read in a geometry object primitive-by-primitive. More information on this is available in the `kgeom_read_primitive()` man page.

The first part of a geometry object should have already been read by a call to `kgeom_start_reading_object()`. That call would have read any information which was needed to read the geometry primitive data. Specifically, the `layout`, `location_dim`, `texture_coord_dim`, and `has_alpha` fields of the object would already have been read in as the first part of the object.

All other information contained within the geometry object is read after the primitives by this call.

The fid internal to the geometry object specifies the file from which the object will be read. In general, this fid is set when the geometry header is read by the `kgeom_read_header()` function.

A.5.23. `kgeom_done_reading()` — *close associated file after reading*

Synopsis

```
int kgeom_done_reading(kgeom_object *object)
```

Input Arguments

object
geometry object to write filename - filename to write header to

Returns

TRUE on success, FALSE otherwise

Description

This function will close the file associated with the geometry object after a read has been completed.

In general, the `kgeom_read_object()` function should be used instead of this function. This function has been made public to provide complete flexibility for reading a geometry object. It should only be used when you wish to read a geometry object primitive-by-primitive. More information on this is available in the `kgeom_read_primitive()` man page.

The `fid` internal to the geometry object specifies the file which will be closed. In general, this `fid` is set when the geometry header was read by the `kgeom_read_header()` function.

This page left intentionally blank

Table of Contents

A. Geometry Data Services	3-1
A.1. Introduction	3-1
A.1.1. The Geometry Data Model	3-2
A.1.1.1. Geometry Primitives	3-2
A.1.1.2. VisiQuest Geometry Format	3-3
A.2. Overview of Geometry Service Primitives	3-3
A.3. The Application Programming Interface (API)	3-7
A.3.1. Geometry Object Functions	3-7
A.3.2. Geometry Primitive Functions	3-8
A.3.3. Primitive List Functions	3-9
A.3.4. Primitives and Data Vectors	3-9
A.3.4.1. Location Data	3-10
A.3.4.2. Color Data	3-10
A.3.4.3. Normal Data	3-10
A.3.4.4. Radius Data	3-11
A.3.4.5. Texture Coordinate Data	3-11
A.3.4.6. Text Data	3-11
A.3.5. Examples	3-11
A.3.5.1. Reading Geometry Data	3-11
A.3.5.2. Writing Geometry Data	3-13
A.4. Geometry Primitives and Associated Attributes	3-17
A.5. Geometry Service Functions	3-26
A.5.1. Object Functions	3-26
A.5.2. <code>kgeom_new_object()</code> — <i>construct a new geometry object</i>	3-26
A.5.3. <code>kgeom_write_object()</code> — <i>write a geometry object</i>	3-28
A.5.4. <code>kgeom_read_object()</code> — <i>read a geometry object</i>	3-29
A.5.5. <code>kgeom_copy_object()</code> — <i>copy a geometry object</i>	3-30
A.5.6. <code>kgeom_blast_object()</code> — <i>free a geometry object</i>	3-31
A.5.7. Primitive Functions	3-31
A.5.8. <code>kgeom_new_primitive()</code> — <i>construct a new geometry primitive</i>	3-31
A.5.9. <code>kgeom_copy_primitive()</code> — <i>copy a geometry primitive</i>	3-34
A.5.10. <code>kgeom_blast_primitive()</code> — <i>destroy a geometry primitive</i>	3-35
A.5.11. Primitive List Functions	3-35
A.5.12. <code>kgeom_add_primitive()</code> — <i>add a primitive to a geometry object</i>	3-35
A.5.13. <code>kgeom_get_primitive()</code> — <i>get a primitive from a geometry object</i>	3-36
A.5.14. <code>kgeom_number_primitives()</code> — <i>count the number of primitives in the given object</i>	3-37
A.5.15. Specialized Reading and Writing Functions	3-38
A.5.16. <code>kgeom_start_writing_object()</code> — <i>write the first part of a geometry object</i>	3-38
A.5.17. <code>kgeom_write_primitive()</code> — <i>write a geometry primitive</i>	3-39
A.5.18. <code>kgeom_finish_writing_object()</code> — <i>write the last part of a geometry object</i>	3-40
A.5.19. <code>kgeom_done_writing()</code> — <i>close associated file after writing</i>	3-41
A.5.20. <code>kgeom_start_reading_object()</code> — <i>read the first part of a geometry object</i>	3-42
A.5.21. <code>kgeom_read_primitive()</code> — <i>read a geometry primitive</i>	3-42
A.5.22. <code>kgeom_finish_reading_object()</code> — <i>read the last part of a geometry object</i>	3-44
A.5.23. <code>kgeom_done_reading()</code> — <i>close associated file after reading</i>	3-44

This page left intentionally blank

Chapter 4

Color Data Services

Chapter 4 - Color Data Services

A. Color Data Services

Color Data Services is designed to support the specific needs of storing information related to color and colormaps. While the other application services are built around their own self-contained data models, Color Data Services is intended to coexist with other data models. The attributes provided through Color Data Services are designed to augment the functionality provided by Polymorphic Data Services and Geometry Data Services. The additional attributes available with the Color Data Services data model allow you to specify how value data vectors should be interpreted by visualization programs as well as providing you with a mechanism for easily creating colormaps and operating on them.

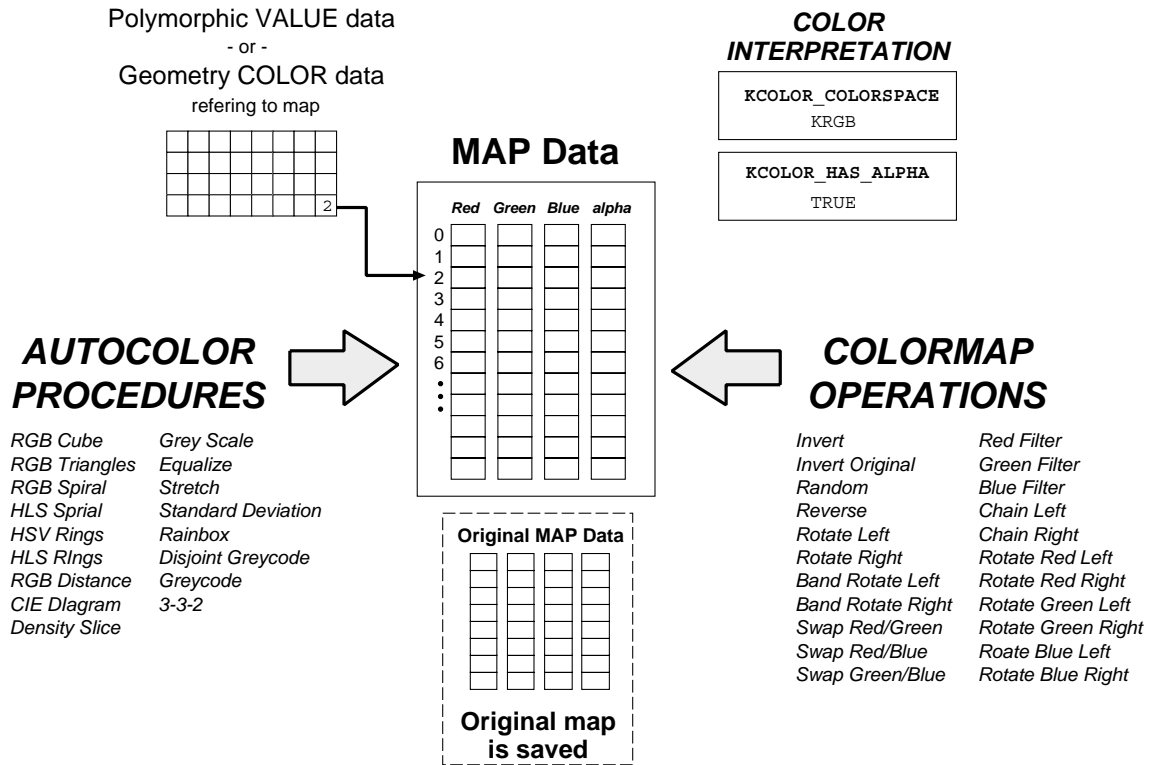


Figure 1: Color Data Services provides you with functionality specific to color data. *Color interpretation* attributes provide information to visual applications indicating how to interpret the color vector information. *Autocolor procedures* and *colormap operations*, which generate new map data and operate on existing map data, are also available. Color Data Services is intended to be used in conjunction with other Data Services, such as Polymorphic Data Services and Geometry Data Services.

A.1. Application Programming Interface (API)

The Application Programming Interface (API) to Color Data Services operates on the same data object abstraction that is used by all Data Services. An abstract data object, represented by the data type *kobject*, is opened by either a Polymorphic Data Services or Geometry Data Services call. This data object is then operated on as it normally would be under that Data Service. When the extra functionality provided by Color Data Services is required, you simply use one of the Color Data Services attribute functions instead of the usual Polymorphic Data Services or Geometry Data Services attribute function calls. Functions are provided for setting, getting, comparing, copying and printing color service attributes.

B. Color Attributes

The following table gives a complete list of the *Color Attributes* currently available. In future releases, the functionality of these attributes is likely to be enhanced and new attributes added.

Color Attributes		
Attribute and Default	Legal Values	Definition
KCOLOR_COLORSPACE int colorspace KNONE	KNONE KGREY KRGB KCMY KYIQ KHSV KHLS KIHS KXYZ KUVW KUCSUVW KUCSSOW KUCSLab KUCSLuv KUSERDEFINED	This attribute indicates the colorspace in which the data in the element vectors of the value segment, or map segment if it is present, are stored. Persistence: permanent
KCOLOR_HAS_ALPHA int has_alpha FALSE		This attribute indicates whether or not an alpha value is contained in the element vectors of the value segment, or map segment if it is present. Persistence: permanent

Color Attributes		
Attribute and Default	Legal Values	Definition
KCOLOR_MAP_AUTOCOLOR int autocolor KORIGINAL	KORIGINAL KRGB_CUBE KRGB_TRIANGLE KRGB_SPIRAL KHLS_SPIRAL KHSV_RINGS KHLS_RINGS KRGB_DISTANCE KCIE_DIAGRAM KDENSITY_SLICE KGREYSCALE KEQUALIZE KSTRETCH KSTDDEV KSA_PSEUDO KRAINBOW KDISJOINT KGREYCODE KMAP332 KRANDOM	<p>This attribute, when called, will create a colormap on an object using the specified autocoloring procedure. There are two types of autocolor mapping schemes available. The first is <i>positional mapping</i>, in which the map values at any position are generated only as a function of that position. The second is <i>data dependent</i> mapping, in which the map values at any position are generated as a function of the value data. The specifics of each autocoloring procedure were explained in earlier sections. The original colormap, if a colormap was present, is saved and can be restored by setting this attribute to KORIGINAL. Note that the original colormap will not be carried along with the data object once it is closed.</p> <p>Persistence: permanent</p>
KCOLOR_MAP_AUTOCOLOR_LIST int num		<p>This attribute returns a list of strings indicating which map autocoloring procedures are available. A number argument indicates how many autocoloring procedures are included in the list.</p> <p>Persistence: permanent</p>

Color Attributes		
Attribute and Default	Legal Values	Definition
KCOLOR_MAP_OPERATION int operation KNONE	KINVERT KINVERT_ORIG KREVERSE KROW_ROTLEFT KROW_ROTRIGHT KCOL_ROTLEFT KCOL_ROTRIGHT KSWAP_REDGREEN KSWAP_REDBLUE KSWAP_GREENBLUE KRED_FILTER KGREEN_FILTER KBLUE_FILTER KCHAIN_ROTLEFT KCHAIN_ROTRIGHT KRED_ROTLEFT KRED_ROTRIGHT KGREEN_ROTLEFT KGREEN_ROTRIGHT KBLUE_ROTLEFT KBLUE_ROTRIGHT	<p>This attribute, when called, will operate on the existing colormap within a data object using the specified colormap operation. A colormap operation will generate a new colormap based on the values in the existing colormap. If an object has no colormap, then setting this attribute will have no effect.</p> <p>The original colormap will be saved before it is altered by any colormap operations and can be restored by setting the attribute KCOLORMAP_OPERATIONS to KORIGINAL. Note that the original colormap will not be carried along with the data object once it is closed.</p>
		Persistence: permanent
KCOLOR_MAP_OPERATION_LIST int num		<p>This attribute returns a list of strings indicating which map operations are available. A number argument indicates how many autocoloring procedures are included in the list.</p>
		Persistence: permanent

C. Color Interpretation

Color Data Services provides two *color interpretation* attributes, KCOLOR_COLORSPACE and KCOLOR_HAS_ALPHA. The colorspace attribute indicates to visualization programs how the color vectors should be interpreted. The *colorspace model* indicates how the value element vectors should be broken down into colors. A colorspace of KNONE or KGREY indicates each element in a value vector represents a single color, while a colorspace of KRGB indicates that every three elements in a value vector represents a single color. If the has alpha attribute is set to true, then it is understood that every color also contains a value indicating an opacity. So, if the colorspace was set to KRGB and has alpha was true, then every four elements in a value vector would represent a single color. Note that these attributes provide extra information *only*. There is nothing to prevent you from setting these attributes to values that do not make sense for your current data. For example, if your element vector size is only two, nothing will prevent you from setting the colorspace model to be

KRGB even though a minimum element vector size of three is required.

D. Autocoloring Procedures and Colormap Operations

Color Data Services provides *autocoloring procedures* and *colormap operations* that can be used to operate on the colormap of a data object. In an autocoloring procedure, the existing colormap is replaced with a pre-defined colormap. In a colormap operation, the existing values in the colormap are modified according to a pre-defined algorithm. For example, the RGB Cube autocoloring procedure replaces the red, green, and blue map columns of the current colormap with the red, green, and blue map columns dictated by the RGB Cube algorithm. In contrast, the Rotate Left colormap operation rotates each of the three columns in the existing colormap to the left. The actual values contained in the map are unchanged.

D.1. Types of Autocoloring Procedures

There are two types of autocoloring procedures. The first type is *positional mapping*, where the map values at any position in map are generated only as a function of that position. Positional mappings have no dependency on the properties of the value data; dependency is only on the physical layout of the color map. RGB cube, RGB triangle, RGB spiral, CIE diagram, and Gray scale are positional color maps. The second type is *data dependent* mapping, in which the map values at any position in the map are generated as a function of the value data. The colormap is generated according to the properties of the value data. For example, the density slice operation maps color according to the histogram of the image pixel values. RGB distance and density slice are data dependent color maps.

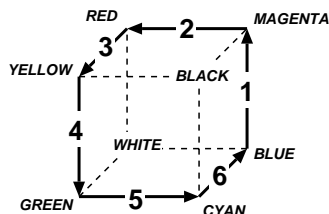
D.2. Available Autocoloring Procedures

In all of the autocoloring procedures described below, the algorithm first finds the minimum and maximum values in the original color map and then "stretches" them so that the resulting color map utilizes the minimum and maximum ranges of each primary. This is done because it allows for better visual discrimination between colors in the enhanced image.

Autocoloring Procedures

Name	Description
KORIGINAL	This autocolor procedure restores the original color map that was contained in the input file when it was first read in.

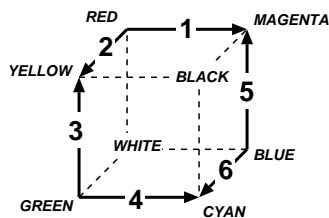
KRGB_CUBE The RGB cube color map is a positional color map which maps the original values to a set of colors that are determined by following the edges of the RGB color cube in a *continuous* manner. The sides of the cube are traced starting from the blue corner in the following order.



RANGE	RED	GREEN	BLUE
1	min->max	min	max
2	max	min	max->min
3	max	min->max	min
4	max->min	max	min
5	min	max	min->max
6	min	max->min	max

The minimum and maximum values in the original color map are found along with the number of colors, N. N is then divided by six which is the number of sides of the RGB cube that are traced. This sets up six ranges with N/6 steps in each range. When the number of colors in the original color map is not a factor of six, the difference is made up in the sixth range. Colors are assigned to these ranges according to the following rules, where min is the minimum primary value (usually 0.0), max is the maximum value (usually 1.0), and min -> max means that the values of that primary are ranged between the minimum and maximum values, or vice versa (max -> min). The minimum and maximum values in the original map are assigned the color blue (0.0,0.0,1.0).

KRGB_TRIANGLE The RGB triangle color map is a positional color map which maps the original values to a set of colors that are determined by following the edges of the RGB color cube in a *disjoint* manner. The sides of the cube are traced starting from the blue corner in the following order.

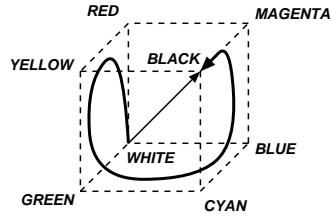


RANGE	RED	GREEN	BLUE
1	max	min	min->max
2	max	min->max	min
3	min->max	max	min
4	min	max	min->max
5	min->max	min	max
6	min	min->max	max

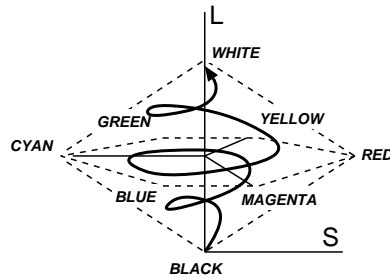
The minimum and maximum values in the original color map are found along with the number of colors, N. N is then divided by six which is the number of sides of the RGB cube that are traced. This sets up six ranges with N/6 steps in each range. When the number of colors in the original color map is not a factor of six, the difference is made up in the sixth range. Colors are assigned to these ranges according to the following rules, where min is the minimum primary value (usually 0.0), max is the maximum value (usually 1.0), and min -> max means that the values of that primary are ranged between the minimum and maximum values, or vice versa (max -> min). The minimum value in the original map is assigned the color red (1.0,0.0,0.0), and the maximum value is assigned the color blue (0.0,0.0,1.0).

Autocoloring Procedures

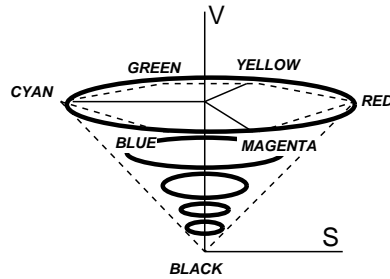
Name	Description
KRGB_SPIRAL	The RGB spiral colormap is a quadratic mapping that maps colors as a spiral which encircles the grey line from black to white.



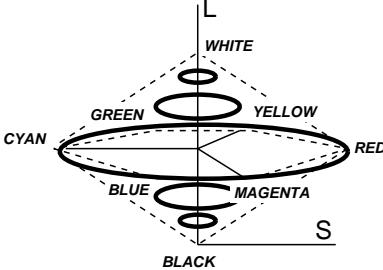
KHLS_SPIRAL	The HLS spiral colormap is constructed by incrementing the lightness, saturation, and hue from 0.0 to 1.0.
-------------	--



KHSV_RINGS	The HSV rings colormap is constructed by incrementing the saturation and value by fixed increments and then varying the hue from 0.0 to 1.0 for each increment. This forms a ring for each increment. The number of rings is dependent on the number of colors being generated.
------------	---



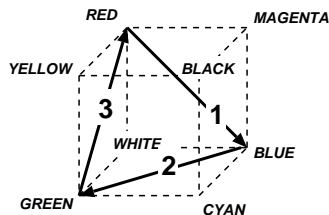
Autocoloring Procedures

Name	Description
KHLS_RINGS	<p>The HLS rings colormap is constructed by incrementing the lightness and saturation by fixed increments and then varying the hue from 0.0 to 1.0. This forms a ring for each increment. The number of rings is dependent on the number of colors being generated.</p> 
KRGB_DISTANCE	<p>The RGB distance colormap is a data dependent color map operation which assigns the red and green components of the color map according to characteristics of the image's histogram, and the blue component according to pixel intensities. The red primary components are determined by calculating a delta variance for each cell of the original color map, and the green components according to a delta mean calculation for each cell. The method of assigning colors is reviewed below.</p> <ol style="list-style-type: none"><li data-bbox="451 810 899 831">1. Calculate mean pixel value of the histogram.<li data-bbox="451 863 1024 884">2. Calculate calculate the average variance of the histogram.<li data-bbox="451 915 760 936">3. Assigning the Red color map: <p data-bbox="451 961 1406 1192">Calculate a delta variance value for each cell in the original color map. This is done by subtracting the average variance of the image's histogram from the number of pixels in the image which have that cell's color This will give a delta variance value for each cell in the original color map. The minimum and maximum delta variance values are found and the minimum is mapped to zero of the red primary and the maximum value is mapped to 1.0 of the red primary. All other cells are assigned red values between 0.0 and 1.0 according to their delta variance value multiplied by a variance scaling factor.</p><li data-bbox="451 1224 786 1245">4. Assigning the Green Color map: <p data-bbox="451 1270 1406 1501">Calculate a delta mean value for each cell in the original color map. This is done by subtracting the mean pixel value of the image's histogram from the number of pixels in the image which have that cell's color This will give a delta mean value for each cell in the original color map. The minimum and maximum delta mean values are found and the minimum is mapped to zero of the green primary and the maximum value is mapped to 1.0 of the green primary. All other cells are assigned green values between 0.0 and 1.0 according to their delta mean value multiplied by a mean scaling factor.</p><li data-bbox="451 1533 773 1554">5. Assigning the Blue Color map: <p data-bbox="451 1579 1406 1675">The blue color map is assigned directly from the intensity values of the image. The lowest intensity value is mapped to 0.0 and the highest intensity value is assigned 1.0. All other intensities are given blue values between 0.0 and 1.0.</p>

Autocoloring Procedures

Name	Description
------	-------------

KCIE_DIAGRAM The CIE Diagram color map is a positional color mapping scheme which maps the original values to a set of colors that are determined by following the edges of a triangle in the RGB color cube in a continuous path. The vertices of this triangle are the red, green, and blue primaries, and the order in which the sides are traced is from red, to blue, to green, to red.

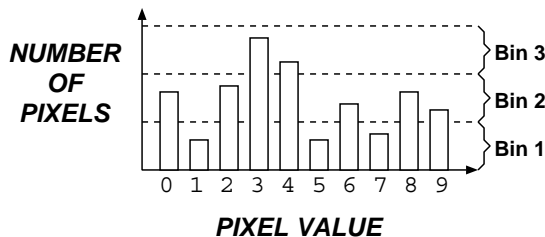


RANGE	RED	GREEN	BLUE
1	max->min	min	min->max
2	min	min->max	max->min
3	min->max	max->min	min

The minimum and maximum values in the original color map are found along with the number of colors, N. N is then divided by three which is the number of sides in the triangle that will be traced. This sets up three ranges with N/3 steps in each range. If the number of colors in the original color map is not a factor of three, the difference is made up in the third range. Colors are assigned to these ranges according to the following rules, where min is the minimum primary value (usually 0.0), max is the maximum value (usually 1.0), and min -> max means that the values of that primary are ranged between the minimum and maximum values, or vice versa (max -> min). The minimum and maximum values in the original map are assigned the color red (1.0,0.0,0.0).

KDENSITY_SLICE The density slice is a data dependent color mapping algorithm which assigns colors according to the distribution of the image's histogram. When the histogram of the image is computed, the maximum number of pixels per cell is found. This is used to determine the bin size for the density slicing operation. Once the bin sizes have been determined, the histogram is searched for cells which fall into each bin, and these pixel values are mapped to colors which vary from green to blue to red (follows the first two legs of the CIE diagram triangle). Blue represents the cell with the fewest number of pixels in the image, and red represents the cell with the highest number of pixels.

A simple example of density slicing is given below. In the example histogram, there are 10 possible pixel values, or cells. The density slice will divide the histogram into 3 slices. Cells which occupy enough pixels in the image to fall into bin 3 will be assigned red, those that fall into bin 2 will be green and those that fall into bin 1 will be blue. Therefore, in this example, pixels 3 and 8 will be red, pixels 0, 4, 5, and 7 will be green, and pixels 1 and 6 will be blue.



KGREYSCALE The grey scale color map is position dependent and maps colors in the original color map to the greys. Since greys are formed by setting red, green, and blue to the same value, the mapping occurs in the RGB cube along the diagonal connecting black (0.0,0.0,0.0) and white (1.0,1.0,1.0).

Autocoloring Procedures

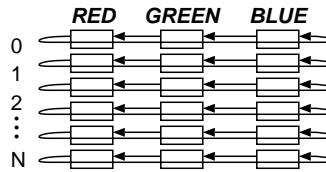
Name	Description
KEQUALIZE	This procedure does a global normalization on the map columns defining the red, green, and blue values of the colormap. Thus, it computes the overall minimum and maximum values over the three map columns, and then normalizes all red, green, and blue values to fall between the overall minimum and the overall maximum.
KSTRETCH	This procedure does a local normalization on the map columns defining the red, green, and blue values of the colormap. It computes the minimums and maximums of the red, green, and blue values separately. It then normalizes the red values between the red minimum and red maximum, the green values between the green minimum and green maximum, and the blue values between the blue minimum and blue maximum.
KSTDDEV	This procedure is not implemented yet.
KSA_PSEUDO	This color map is based on an article found in Scientific American. The color map is designed to convert a grey scale image and convert it to a color image of the same intensity. The algorithm takes each entry and maps it to a corresponding red, green, blue value that has been chosen more for it's aesthetics than it's quantitative value. The article was discovered by Joe Fogler who typed in the value tables and added a few corrections to make the color map more aesthetically pleasing.
KRAINBOW	The rainbow color map is position dependent and maps colors in the original color map to a rainbow by traversing through HSV space by fixing the the value and saturation to 1.0 and varying the hue between 0.0 and 1.0.
KDISJOINT	This colormap is constructed from a 3 bit disjoint code, where each successive value changes by two bits. Each 3 bit value represents an RGB value. A bit of 1 implies that the maximum intensity is used for that color and a bit of 0 implies that the color is not used. The final map is then interpolated out from the greyscale map to the provide the desired number of colors.
KGREYCODE	This colormap is constructed from a 3 bit grey code, where each successive value changes by a single bit. Each 3 bit value represents an RGB value. A bit of 1 implies that the maximum intensity is used for that color and a bit of 0 implies that the color is not used. The final map is then interpolated out from the greyscale map to the provide the desired number of colors.
KMAP332	This is a predefined colormap that uses three bits of primary for red, three bits of primary for green, and two bit primary for blue. It is typically used for quickly mapping RGB images for 8-bit visualization.
KRANDOM	This creates a color map with totally random values. The routine assigns random values (0.0 - 1.0) to each red, green, blue primary. The uniform random function <code>w(kurng())</code> is used to generate the random numbers.

D.3. Available Colormap Operations

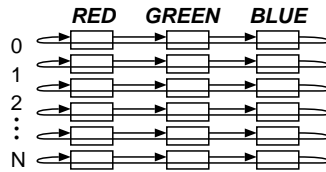
Recall that a colormap operation differs from an autocolor procedure in that it uses the values in the existing colormap to define a new colormap, rather than simply replacing the existing colormap with a predefined colormap. There are a number of colormap operations that may be used. The available colormap operations are described in this section.

Colormap Operations

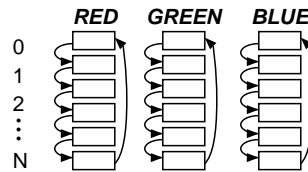
Name	Description
KINVERT	Selecting this button will invert whichever color map is currently being displayed. This is done by subtracting each red, green, blue value from 1.0. (red, green, blue) -> (1.0-red, 1.0-green, 1.0-blue).
KINVERT_ORIG	This assigns to an image a color map that is the original color map with inverted values.
KREVERSE	This colormap operation reverses each of the columns in the colormap. Thus, for colormaps of height N, that the map values that used to be at 0 are now at N-1, the values that used to be at N-1 are now at zero; all map values between 0 and N-1 similarly have their positions switched.
KROW_ROTLEFT	This colormap operation does a rotation on each row of the map. For each RGB color, the green color value will be moved to the red position, the blue value will be moved to the green position, and the red value will be moved to the blue position.



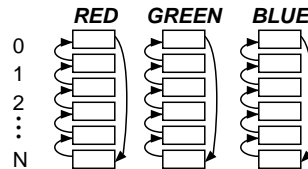
KROW_ROTRIGHT	This colormap operation does a rotation on each row of the map. For each RGB color, the green color value will be moved to the red position, the blue value will be moved to the green position, and the red value will be moved to the blue position.
---------------	--



KCOL_ROTLEFT	This colormap operation does a rotation on the height of each of the map columns. Thus, for colormaps of height N, the map value at 0 is moved to 1, the map value at 1 is moved to 2, and so on; the map value at N-1 is moved to 0.
--------------	---



KCOL_ROTRIGHT	This colormap operation does a rotation on the height of each of the map columns. Thus, for colormaps of height N, the map value at N-1 is moved to N-2, the map value at N-2 is moved to N-3, and so on; the map value at 0 is moved to N-1.
---------------	---



Colormap Operations

Name	Description
KSWAP_REDGREEN	<p>This colormap operation swaps the positions of the red column and the green column.</p> <div style="text-align: center; padding: 10px;"> </div>
KSWAP_REDBLUE	<p>This colormap operation swaps the positions of the red column and the blue column.</p> <div style="text-align: center; padding: 10px;"> </div>
KSWAP_GREENBLUE	<p>This colormap operation swaps the positions of the green column and the blue column.</p> <div style="text-align: center; padding: 10px;"> </div>
KRED_FILTER	<p>This colormap operation sets all the values in the red map column to zero, thus removing the red element from each of the map values.</p> <div style="text-align: center; padding: 10px;"> </div>
KGREEN_FILTER	<p>This colormap operation sets all the values in the green map column to zero, thus removing the green element from each of the map values.</p> <div style="text-align: center; padding: 10px;"> </div>

Colormap Operations

Name	Description																								
KBLUE_FILTER	<p>This colormap operation sets all the values in the blue map column to zero, thus removing the blue element from each of the map values.</p> <div style="text-align: center; margin-top: 10px;"> <table style="border-collapse: collapse; margin: auto;"> <thead> <tr> <th style="padding: 5px;"></th> <th style="padding: 5px;"><i>RED</i></th> <th style="padding: 5px;"><i>GREEN</i></th> <th style="padding: 5px;"><i>BLUE</i></th> </tr> </thead> <tbody> <tr> <td style="padding: 5px;">0</td> <td style="padding: 5px; border: 1px solid black; width: 20px; height: 15px;"></td> <td style="padding: 5px; border: 1px solid black; width: 20px; height: 15px;"></td> <td style="padding: 5px; border: 1px solid black; width: 20px; height: 15px; text-align: center;">0</td> </tr> <tr> <td style="padding: 5px;">1</td> <td style="padding: 5px; border: 1px solid black; width: 20px; height: 15px;"></td> <td style="padding: 5px; border: 1px solid black; width: 20px; height: 15px;"></td> <td style="padding: 5px; border: 1px solid black; width: 20px; height: 15px; text-align: center;">0</td> </tr> <tr> <td style="padding: 5px;">2</td> <td style="padding: 5px; border: 1px solid black; width: 20px; height: 15px;"></td> <td style="padding: 5px; border: 1px solid black; width: 20px; height: 15px;"></td> <td style="padding: 5px; border: 1px solid black; width: 20px; height: 15px; text-align: center;">0</td> </tr> <tr> <td style="padding: 5px;">⋮</td> <td style="padding: 5px; border: 1px solid black; width: 20px; height: 15px;"></td> <td style="padding: 5px; border: 1px solid black; width: 20px; height: 15px;"></td> <td style="padding: 5px; border: 1px solid black; width: 20px; height: 15px; text-align: center;">0</td> </tr> <tr> <td style="padding: 5px;">N</td> <td style="padding: 5px; border: 1px solid black; width: 20px; height: 15px;"></td> <td style="padding: 5px; border: 1px solid black; width: 20px; height: 15px;"></td> <td style="padding: 5px; border: 1px solid black; width: 20px; height: 15px; text-align: center;">0</td> </tr> </tbody> </table> </div>		<i>RED</i>	<i>GREEN</i>	<i>BLUE</i>	0			0	1			0	2			0	⋮			0	N			0
	<i>RED</i>	<i>GREEN</i>	<i>BLUE</i>																						
0			0																						
1			0																						
2			0																						
⋮			0																						
N			0																						
KCHAIN_ROTLEFT	<p>This colormap operation does a rotation on the height of each map column, where each map value is moved up one position, and the 0th map value of each column is moved to the last position of the next column. Thus, for colormaps of height N, the red map value at N-1 is moved to N-2, the red map value at N-2 is moved to N-3, and so on; the red map value at position 0 is moved to the green map column, at position N-1.</p> <div style="text-align: center; margin-top: 10px;"> <p>The diagram shows three vertical columns of boxes labeled RED, GREEN, and BLUE. To the left of the boxes are indices 0, 1, 2, ⋮, N. Curved arrows indicate the rotation: in the RED column, arrows point from index i to i-1 (for i > 0) and from index 0 to index N-1. Similar arrows are shown for the GREEN and BLUE columns, indicating a cyclic shift of one position up in each column.</p> </div>																								
KCHAIN_ROTRIGHT	<p>This colormap operation does a rotation on the height of each map column, where each map value is moved back one position, and the last map value of each column is moved to the first position of the next column. Thus, for colormaps of height N, the red map value at 0 is moved to 1, the red map value at 1 is moved to 2, and so on; the red map value at N-1 is moved to the 0th position of the green column.</p> <div style="text-align: center; margin-top: 10px;"> <p>The diagram shows three vertical columns of boxes labeled RED, GREEN, and BLUE. To the left of the boxes are indices 0, 1, 2, ⋮, N. Curved arrows indicate the rotation: in the RED column, arrows point from index i to i+1 (for i < N-1) and from index N-1 to index 0. Similar arrows are shown for the GREEN and BLUE columns, indicating a cyclic shift of one position down in each column.</p> </div>																								
KRED_ROTLEFT	<p>This colormap operation does a rotation on the height of the red map column. Thus, for colormaps of height N, the red map value at N-1 is moved to N-2, the red map value at N-2 is moved to N-3, and so on; the red map value at 0 is moved to N-1.</p> <div style="text-align: center; margin-top: 10px;"> <p>The diagram shows three vertical columns of boxes labeled RED, GREEN, and BLUE. To the left of the boxes are indices 0, 1, 2, ⋮, N. Curved arrows are only shown within the RED column, indicating a cyclic shift of one position up (from i to i-1, and from 0 to N-1). The GREEN and BLUE columns are empty.</p> </div>																								

Colormap Operations

Name	Description
KRED_ROTRIGHT	<p>This colormap operation does a rotation on the height of the red map column. Thus, for colormaps of height N, the red map value at 0 is moved to 1, the red map value at 1 is moved to 2, and so on; the red map value at $N-1$ is moved to 0.</p> <div style="text-align: center; margin: 10px 0;"> </div>
KGREEN_ROTLEFT	<p>This colormap operation does a rotation on the height of the green map column. Thus, for colormaps of height N, the green map value at $N-1$ is moved to $N-2$, the green map value at $N-2$ is moved to $N-3$, and so on; the green map value at 0 is moved to $N-1$.</p> <div style="text-align: center; margin: 10px 0;"> </div>
KGREEN_ROTRIGHT	<p>This colormap operation does a rotation on the height of the green map column. Thus, for colormaps of height N, the green map value at 0 is moved to 1, the green map value at 1 is moved to 2, and so on; the green map value at $N-1$ is moved to 0.</p> <div style="text-align: center; margin: 10px 0;"> </div>
KBLUE_ROTLEFT	<p>This colormap operation does a rotation on the height of the blue map column. Thus, for colormaps of height N, the blue map value at $N-1$ is moved to $N-2$, the blue map value at $N-2$ is moved to $N-3$, and so on; the blue map value at 0 is moved to $N-1$.</p> <div style="text-align: center; margin: 10px 0;"> </div>

Colormap Operations

Name	Description
KBLUE_ROTRIGHT	This colormap operation does a rotation on the height of each of only the blue map column. Thus, for colormaps of height N, the blue map value at 0 is moved to 1, the blue map value at 1 is moved to 2, and so on; the blue map value at N-1 is moved to 0.

E. Color Data Services Functions

All the *functions* available for Color Data Services are used for accessing color attributes from a data object.

- *kcolor_set_attribute()* - set the value of a color attribute in a data object.
- *kcolor_set_attributes()* - set multiple color attributes in a data object.
- *kcolor_get_attribute()* - get the values of a color attribute from a data object.
- *kcolor_get_attributes()* - get multiple color attributes from a data object.
- *kcolor_match_attribute()* - compare a color attribute between two data objects.
- *kcolor_match_attributes()* - compare multiple attributes between two objects.
- *kcolor_copy_attribute()* - copy a color attribute from one data object to another.
- *kcolor_copy_attributes()* - copy multiple attributes from one object to another.
- *kcolor_query_attribute()* - query characteristics of a color attribute.
- *kcolor_print_attribute()* - print the value of a color attribute from a data object.
- *kcolor_gamut_object()* - perform color quantization of 1..4 plane images

E.1. *kcolor_set_attribute()* — *set the value of a color attribute in a data object.*

Synopsis

```
int kcolor_set_attribute(
    kobject object,
    char *attribute,
    kvalist)
```

Input Arguments

`object`
the data object into which the attribute's value will be assigned

`attribute`
the attribute to set

`kvalist`
a variable argument list that contains the values that will be assigned to the different components

associated with that attribute.

The variable argument list takes the form:

```
ATTRIBUTE_NAME, value1 [, value2, ...]
```

The number of value arguments in the variable argument list corresponds to the number of arguments needed to set the attribute.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to assign the value of a color attribute to a data object.

This color service function should be used in conjunction with other application services such as polymorphic data services and geometry data services. This function will work on data objects opened or created with either of those services.

Attributes are set by passing in the attribute name along with the value or variable containing the value to assign to the attribute.

The following example illustrates the use of the set attribute call to assign two different color attributes. The define `KRGB` could have been passed in directly to set the colorspace attribute.

```
int    colorspace = KRGB;

kcolor_set_attribute(object,
                    KCOLOR_COLORSPACE, colorspace);,
kcolor_set_attribute(object,
                    KCOLOR_MAP_OPERATION, KROW_LEFT);
```

A complete list of color attributes can be found in Chapter 4 of Programming Services Volume II.

Restrictions

Calling this function with an incorrect number of arguments in the variable argument list will not cause any compiler errors, but will often generate a segmentation fault.

E.2. `kcolor_set_attributes()` — *set multiple color attributes in a data object.*

Synopsis

```
int kcolor_set_attributes(  
    kobject object,  
    kvalist)
```

Input Arguments

`object`
the data object into which the values of the attributes will be assigned

`kvalist`
NULL terminated variable argument list which contains a list of attributes, each attribute followed by the values to assign to that attribute.

The variable argument list takes the form:

```
ATTRIBUTE_NAME1, value1 [, value2, ...],  
ATTRIBUTE_NAME2, value1 [, value2, ...],  
..., NULL
```

The number of value arguments in the variable argument list for each attribute depends on the attribute. The NULL at the end of the variable argument list serves as a flag indicating the end of the list.

Be careful not to forget the NULL at the end of the list. This is a common programming error which unfortunately will not generate any compiler warnings.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to assign the values of an arbitrary number of color attributes to a data object.

This color service function should be used in conjunction with other application services such as polymorphic data services and geometry data services. This function will work on data objects opened or created with either of those services.

Attributes are set by passing in the attribute name along with the value or variable containing the value to assign to the attribute.

The following example illustrates the use of a single set attributes call to assign two different color attributes. The define `KRGB` could have been passed in directly to set the colorspace attribute.

```

int    colorspace = KRGB;

kcolor_set_attributes(object,
                    KCOLOR_COLORSPACE,    colorspace,
                    KCOLOR_MAP_OPERATION, KROW_LEFT,
                    NULL);

```

A complete list of color attributes can be found in Chapter 4 of Programming Services Volume II.

Restrictions

Calling this function with an incorrect number of arguments in the variable argument list will not cause any compiler errors, but will often generate a segmentation fault.

E.3. `kcolor_get_attribute()` — *get the values of a color attribute from a data object.*

Synopsis

```

int kcolor_get_attribute(
    kobject object,
    char *attribute,
    kvalist)

```

Input Arguments

`object`
the data object from which the attribute's value will be retrieved

`attribute`
the attribute to get

`kvalist`
a variable argument list that contains the addresses of variables which will be used to return the different components associated with that attribute.

The variable argument list takes the form:

```

ATTRIBUTE_NAME, &value1 [, &value2, ...]

```

The number of value arguments in the variable argument list corresponds to the number of arguments needed to retrieve the attribute.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to retrieve the value of a color attribute from a data object.

This color service function should be used in conjunction with other application services such as polymorphic data services and geometry data services. This function will work on data objects opened or created with either of those services.

Attributes are retrieved by passing in the address of a variable by which the attribute can be returned. Note that any array attributes, such as strings, which are retrieved should not be altered or freed. The pointer returned points to the actual internal storage array. A copy should be made if the values need to be changed.

The following example illustrates the use of the `get` attribute call to retrieve two different color attributes.

```
char **autocolor_list;
int    num;
int    colorspace;

kcolor_get_attribute(object,
                    KCOLOR_AUTOCOLOR_LIST, &list, &num);
kcolor_get_attribute(object,
                    KCOLOR_COLORSPACE, &colorspace);
```

A complete list of color attributes can be found in Chapter 4 of Programming Services Volume II.

Restrictions

Calling this function with an incorrect number of arguments in the variable argument list will not cause any compiler errors, but will often generate a segmentation fault.

E.4. `kcolor_get_attributes()` — *get multiple color attributes from a data object.*

Synopsis

```
int kcolor_get_attributes(
    kobject object,
    kvalist)
```

Input Arguments

`object`
the data object from which the values of the attributes will be retrieved

Output Arguments

`kvalist`

NULL terminated variable argument list which contains a list of attributes, each attribute followed by the addresses of variables which will be used to return the different components associated with that attribute.

The variable argument list takes the form:

```
ATTRIBUTE_NAME1, &value1 [, &value2, ...],  
ATTRIBUTE_NAME2, &value1 [, &value2, ...],  
..., NULL
```

The number of value arguments in the variable argument list for each attribute depends on the attribute. The NULL at the end of the variable argument list serves as a flag indicating the end of the list.

Be careful not to forget the NULL at the end of the list. This is a common programming error which unfortunately will not generate any compiler warnings.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to retrieve the values of an arbitrary number of attributes from a data object.

This color service function should be used in conjunction with other application services such as polymorphic data services and geometry data services. This function will work on data objects opened or created with either of those services.

Attributes are retrieved by passing in the address of a variable by which the attribute can be returned. Note that any array attributes, such as strings, which are retrieved should not be altered or freed. The pointer returned points to the actual internal storage array. A copy should be made if the values need to be changed.

The following example illustrates the use of a single get attributes call to retrieve two different color attributes.

```
char **autocolor_list;  
int    num;  
int    colorspace;  
  
kcolor_get_attributes(object,  
                     KCOLOR_AUTOCOLOR_LIST, &list, &num,  
                     KCOLOR_COLORSPACE, &colorspace,  
                     NULL);
```

A complete list of color attributes can be found in Chapter 4 of Programming Services Volume II.

Restrictions

Calling this function with an incorrect number of arguments in the variable argument list will not cause any compiler errors, but will often generate a segmentation fault.

E.5. `kcolor_match_attribute()` — *compare a color attribute between two data objects.*

Synopsis

```
int kcolor_match_attribute(  
    kobject object1,  
    kobject object2,  
    char *attribute)
```

Input Arguments

`object1`
the first data object containing the attribute to be compared

`object2`
the second data object containing the attribute to be compared

`attribute`
the attribute to be compared

Returns

TRUE (1) if the attribute matches, FALSE (0) otherwise

Description

This function is used to compare the value of a color attribute between two data objects.

This color service function should be used in conjunction with other application services such as polymorphic data services and geometry data services. This function will work on data objects opened or created with either of those services.

If the value of the attribute in both objects is the same, then this function will return TRUE. If the values are different, then this function will return FALSE.

The following example illustrates the use of the match attribute call to compare two different color attributes.

```
if (kcolor_match_attribute(obj1, obj2, KCOLOR_COLORSPACE))
```

```
kprintf("colorspace is the same in both objects");
if (kcolor_match_attribute(obj1, obj2, KCOLOR_MAP_OPERATION))
kprintf("colormap operation is the same in both objects");
```

A complete list of color attributes can be found in Chapter 4 of Programming Services Volume II.

E.6. kcolor_match_attributes() — *compare multiple attributes between two objects.*

Synopsis

```
int kcolor_match_attributes(
    kobject object1,
    kobject object2,
    kvalist)
```

Input Arguments

`object1`

the first data object containing the attributes to be compared

`object2`

the second data object containing the attributes to be compared

`kvalist`

NULL terminated variable argument list which contains a list of attributes to be compared.

The variable argument list takes the form:

```
ATTRIBUTE_NAME1,
ATTRIBUTE_NAME2,
..., NULL
```

Returns

TRUE (1) if all listed attributes match, FALSE (0) otherwise

Description

This function is used to compare the values of an arbitrary number of color attributes between two data objects.

This color service function should be used in conjunction with other application services such as polymorphic data services and geometry data services. This function will work on data objects opened or created with either of those services.

If the value of all attributes in both objects are the same, then this function will return TRUE. If any of the values are different, then this function will return FALSE.

The following example illustrates the use of the match attributes call to compare two different color attributes.

```
if (kcolor_match_attributes(obj1, obj2,
                           KCOLOR_COLORSPACE,
                           KCOLOR_MAP_OPERATION,
                           NULL))
    kprintf("colorspace and colormap operation are the same "
           "in both objects");
```

A complete list of color attributes can be found in Chapter 4 of Programming Services Volume II.

Restrictions

Calling this function and forgetting to NULL terminate the variable argument list will not cause any compiler errors, but will often generate a segmentation fault.

E.7. `kcolor_copy_attribute()` — *copy a color attribute from one data object to another.*

Synopsis

```
int kcolor_copy_attribute(
    kobject object1,
    kobject object2,
    char *attribute)
```

Input Arguments

`object1`
the object to use as the source for the copy

`object2`
the object to use as the destination for the copy

`attribute`
the attribute to be compared

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to copy the value of a color attribute from one data object to another.

This color service function should be used in conjunction with other application services such as polymorphic data services and geometry data services. This function will work on data objects opened or created with either of those services.

The following example illustrates the use of the copy attribute call to copy two different color attributes.

```
kcolor_copy_attribute(obj1, obj2, KCOLOR_COLORSPACE);,  
kcolor_copy_attribute(obj1, obj2, KCOLOR_MAP_OPERATION);,
```

A complete list of color attributes can be found in Chapter 4 of Programming Services Volume II.

E.8. kcolor_copy_attributes() — *copy multiple attributes from one object to another.*

Synopsis

```
int kcolor_copy_attributes(  
    kobject object1,  
    kobject object2,  
    kvalist)
```

Input Arguments

`object1`
the object to use as the source for the copy

`object2`
the object to use as the destination for the copy

`kvalist`
NULL terminated variable argument list which contains a list of attributes to be copied.

The variable argument list takes the form:

```
ATTRIBUTE_NAME1,  
ATTRIBUTE_NAME2,  
..., NULL
```


Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to copy the values of an arbitrary number of color attributes from one data object to another.

This color service function should be used in conjunction with other application services such as polymorphic data services and geometry data services. This function will work on data objects opened or created with either of those services.

The following example illustrates the use of the copy attributes call to compare two different color attributes.

```
kcolor_copy_attributes(obj1, obj2
                      KCOLOR_COLORSPACE,
                      KCOLOR_MAP_OPERATION,
                      NULL);
```

A complete list of color attributes can be found in Chapter 4 of Programming Services Volume II.

Restrictions

Calling this function and forgetting to NULL terminate the variable argument list will not cause any compiler errors, but will often generate a segmentation fault.

E.9. kcolor_query_attribute() — *query characteristics of a color attribute.*

Synopsis

```
int kcolor_query_attribute(
    kobject object,
    char *attribute,
    int *num_args,
    int *arg_size,
    int *data_type,
    int *permanent)
```

Input Arguments

object
the data object to be queried for the existence of the named attribute

attribute

the attribute to query

Output Arguments

`num_args`
number of arguments

`arg_size`
size of the arguments, or NULL

`data_type`
data type of the attribute

`permanent`
TRUE if the attribute is stored with the object, FALSE if the attribute is transient

Returns

TRUE (1) if attribute exists, FALSE (0) otherwise

Description

This function is used to query characteristics of a color attribute from a data object.

This color service function should be used in conjunction with other application services such as polymorphic data services and geometry data services. This function will work on data objects opened or created with either of those services.

The following example illustrates the use of the query attribute call to determine the data type of the colorspace attribute.

```
int    data_type;

kcolor_query_attributes(object, KCOLOR_COLORSPACE,
                       NULL, NULL, &data_type, NULL);
```

A complete list of color attributes can be found in Chapter 4 of Programming Services Volume II.

E.10. `kcolor_print_attribute()` — *print the value of a color attribute from a data object.*

Synopsis

```
int kcolor_print_attribute(
    kobject object,
    char    *attribute,
    kfile   *printfile)
```

Input Arguments

`object`

the data object containing the attribute to be printed

`attribute`

the attribute to print

`printfile`

a file or transport pointer opened with `kfopen`. `kstdout` and `kstderr` may be used to print to standard out and standard error.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to print the value of a color attribute from a data object to an output file.

This color service function should be used in conjunction with other application services such as polymorphic data services and geometry data services. This function will work on data objects opened or created with either of those services.

The following example illustrates the use of the `print` attribute call to print the `colorspace` attribute to the output file "outputfile".

```
kfile *outfile = kfopen("outputfile");  
  
kcolor_print_attributes(object, KCOLOR_COLORSPACE, outfile);
```

A complete list of color attributes can be found in Chapter 4 of Programming Services Volume II.

E.11. `kcolor_gamut_object()` — *perform color quantization of 1..4 plane images*

Synopsis

```
int kcolor_gamut_object (  
    kobject src,  
    int ncolors,  
    int bits,  
    double fraction,  
    kobject dest )
```

Input Arguments

`src`

object to be color quantized

`ncolors`
desired number of colors for result image, should be (1..65535)

`bits`
number of bits of resolution to keep from each color plane. Should be(1..8). Higher values give better results but take longer and require more memory.

`fraction`
fraction of color splits to be base on the span of the color space versus the population count in the color space. 1.0 means split only on span. 0.0 means split only on population. 0.0 is effectively a popularity contest. You will usually get best results somewhere around 0.5.

Output Arguments

`dest`
object to hold quantized result

Returns

TRUE (1) on success, FALSE (0) on failure

Description

`kcolor_gamut_object` uses a variation of Paul Heckbert's median cut algorithm to perform color quantization of one, two, three, or four plane images producing a single-plane image with color map.

The quantization is performed by isolating clusters of "neighboring" color vectors in a four dimensional histogram, with each axis being one of the color components. The clusters are obtained using a modified version of Heckbert's median cut. The true colors are then matched to the closest cluster, and the input vector is then re-mapped to an n-color pseudo color image.

To keep the histogram from becoming exceedingly large (max of around 2^{24} bytes), one may need to quantize the grey levels of the input bands to less than 8 bits. 6 bits (64 levels) gives results that are reasonable in a short amount of time. The number of bits that are kept is called the color precision, which can be specified at execution time. The general tradeoff is that smaller precision is faster and takes less memory, but it looks worse too. High precision takes longer and great gobs of memory, but looks decent, provided that a reasonable number (say 128 or more) colors is specified. The execution time is very dependent on the image statistics. In general, a small number of colors is faster than a large number of colors. In either case, if the image has good spatial color coherence, execution time is greatly reduced.

The allocation fraction controls how large areas of nearly the same color are handled. An allocation fraction of 0.0 will cause the large areas to be broken into as many colors as possible with the largest areas of a particular color range being broken first. An allocation fraction of 1.0 will attempt to preserve the detail in the image by preserving the color range of all parts of the image at the expense of smooth coloring of the larger areas. An allocation fraction of around 0.2 to 0.5 gives very good results on most images.

If the input image contains less than the number of colors requested then the output image will contain only the number of colors present in the input image. The color map will contain the number of entries requested (meaning all colors in the image) with any extra entries zero padded.

Multiple plane images are processed by quantizing each plane independently, generating a corresponding plane of colors in the map. Thus an input object with $(w,h,d,t,e)=(512,480,10,10,4)$ will result in an output object with a value segment with dimensions $(512,480,10,10,1)$ and a map segment with dimensions $(4,10,10,10,1)$.

This page left intentionally blank

Table of Contents

A. Color Data Services	4-1
A.1. Application Programming Interface (API)	4-2
B. Color Attributes	4-2
C. Color Interpretation	4-4
D. Autocoloring Procedures and Colormap Operations	4-5
D.1. Types of Autocoloring Procedures	4-5
D.2. Available Autocoloring Procedures	4-5
D.3. Available Colormap Operations	4-10
E. Color Data Services Functions	4-15
E.1. <code>kcolor_set_attribute()</code> — <i>set the value of a color attribute in a data object.</i>	4-15
E.2. <code>kcolor_set_attributes()</code> — <i>set multiple color attributes in a data object.</i>	4-17
E.3. <code>kcolor_get_attribute()</code> — <i>get the values of a color attribute from a data object.</i>	4-18
E.4. <code>kcolor_get_attributes()</code> — <i>get multiple color attributes from a data object.</i>	4-19
E.5. <code>kcolor_match_attribute()</code> — <i>compare a color attribute between two data objects.</i>	4-21
E.6. <code>kcolor_match_attributes()</code> — <i>compare multiple attributes between two objects.</i>	4-22
E.7. <code>kcolor_copy_attribute()</code> — <i>copy a color attribute from one data object to another.</i>	4-23
E.8. <code>kcolor_copy_attributes()</code> — <i>copy multiple attributes from one object to another.</i>	4-24
E.9. <code>kcolor_query_attribute()</code> — <i>query characteristics of a color attribute.</i>	4-25
E.10. <code>kcolor_print_attribute()</code> — <i>print the value of a color attribute from a data object.</i>	4-26
E.11. <code>kcolor_gamut_object()</code> — <i>perform color quantization of 1..4 plane images</i>	4-27

This page left intentionally blank

Chapter 5

Data Management Services

Chapter 5 - Data Management Services

A. Introduction

Data Management Services is the infrastructure that provides data abstraction, large data set processing capability, attribute management, file format independence, and presentation facilities to the application services described in preceding chapters. *Data Management Services* is not intended to be used as a general purpose data access facility; that is the purpose of *Polymorphic Data Services*, described in Chapter 2.

Unlike application services, *Data Management Services* provides no management of the *Polymorphic Data Model*. It is the purpose of *Data Management Services* to provide a basic framework for developing application services and to support the implementation of application-specific data models. Figure 1 illustrates the association between application services and this *Data Service*.

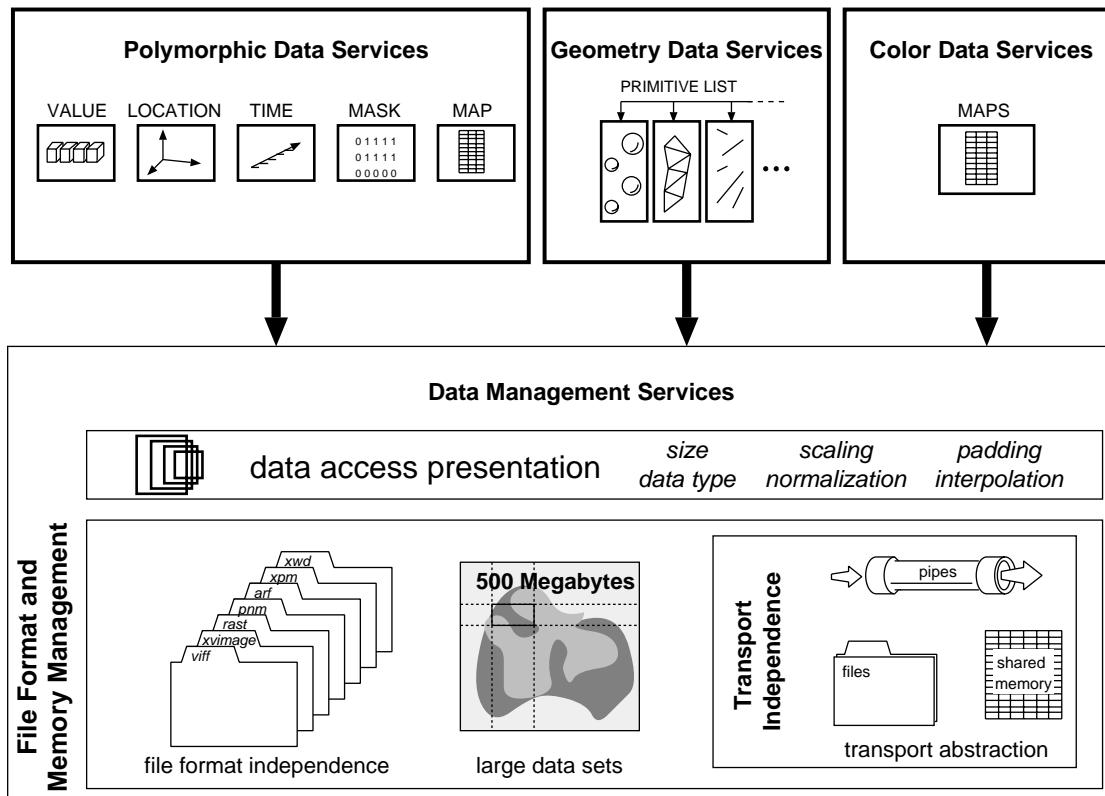


Figure 1: Data Management Services is an infrastructure service provided primarily to facilitate development of application services.

An application service provides a data model that imposes an interpretation on specific segments and provides domain specific functionality to better facilitate a particular style of interaction with a data object. *Data Management Services*, on the other hand, does *not* attempt to enforce any interpretation on a data object whatsoever. Furthermore, no association is enforced between any two segments. For example, the *Polymorphic Data Model* dictates that the *mask* and the *value* segments will be the same size, and *Polymorphic Data Services*

enforces this policy. Data Management Services will not enforce this policy or any other policy that restricts interaction with the data object.

Data Management Services implements three basic constructs: data objects, segments, and attributes. An attribute can be associated with either an object or a segment. Segments exist as part of an object. Figure 2 illustrates the organization of a data object.

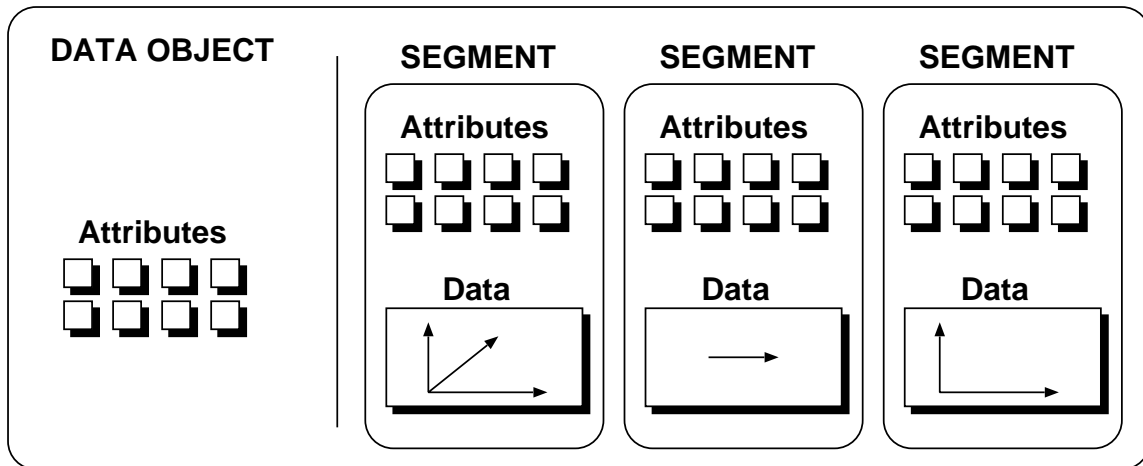


Figure 2: Data objects are implemented as a complex structure that contains any number of segments and any number of attributes. Segments can similarly contain any number of segments.

B. Presentation of Data †

Data Management Services has the ability to present the data stored within a data object in a variety of ways. Data can be cast, resized, normalized, scaled, or re-oriented on access. The API to this functionality is provided by a number of attributes. By setting the appropriate attributes, Data Management Services will return the data in the form that is most convenient to process. In order to understand how the presentation attributes are used, it is necessary to understand how the data object is divided into a presentation layer and physical layer.

A data object can be thought of in terms of two layers: a *presentation layer* and a *physical layer*. Attributes at the physical layer typically describe the actual stored characteristics of the data. Attributes at the presentation layer typically dictate how the data is to be accessed. For example, there is a physical data type attribute indicating the data type in which the data is actually stored, and a presentation data type attribute indicating in which data type the data should be presented. If the presentation data type is set to integer, while the physical data type is set to short, then the data will be cast from short to integer on retrieval and from integer to short on storage.

† Much of this discussion is similar to text in Chapters 1 and 2. It is repeated here in the context of Data Management Services, but aside from the differences in API, the functionality is nearly identical. One new section that was not presented in other chapters is related to index order manipulation.

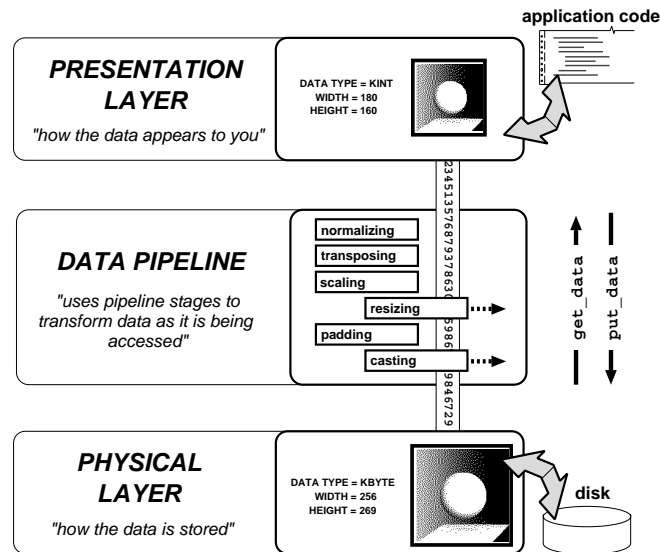


Figure 3: A data object can be thought of as having two layers, a *presentation layer* and a *physical layer*. Attributes at the physical layer determine the storage characteristics of the data, such as its size and data type. Attributes at the presentation layer determine the presentation characteristics of the data. On access, data is passed through a data pipeline which transforms the data according to the presentation attributes. Each presentation attribute corresponds to a stage in the data pipeline; only the necessary stages are invoked on data access.

The following sections outline the different mechanisms that are available for customizing data access.

B.1. Casting

The *casting* feature provided by Data Management Services is used to change the data type from the type stored to another data type that is more suitable for processing. This process is nearly automatic. You have to set the data type of the segment that is being operated on to the desired processing data type with a call to `kdms_set_attribute()` or `kdms_set_attributes()`. Afterward, all data retrieved with `kdms_get_data()` will be returned to the user in the data type specified, regardless of the stored data type. If operating on an output object, then setting the data type of the output object to something different from the stored data type informs Data Management Services that any data being written with `kdms_put_data()` will be given in the specified data type, but should be cast before being written out.

The casting feature is performed via the ANSI C cast operation. Since ANSI C does not dictate the behavior of certain lossy cast operations, such as signed information cast to an unsigned data type, the behavior of this operation in certain circumstances can vary from platform to platform.

B.2. Scaling and Normalization

Scaling and normalization are two activities that alter the range of data when presented to the user. These operations are often necessary when processing data where an algorithm operates better on a limited range of data. After indicating that scaling or normalization is to occur, any call to `kdms_get_data()` will cause the

range of the data to be altered before it is returned to the calling program. The attribute `KDMS_SCALING` determines what kind of range alteration is to occur. The default value for this attribute is `KNONE`, which indicates that no scaling whatsoever is to occur. Other legal values for this attribute are `KSCALE` and `KNORMALIZE`.

When the scaling attribute is set to `KSCALE`, then the range of the data is controlled by two attributes: `KDMS_SCALE_FACTOR` and `KDMS_SCALE_OFFSET`. The range change is computed by applying the scale factor first to each data point, then adding the scale offset.

When the scaling attribute is set to `KNORMALIZE`, then the range of the data is controlled by two other attributes: `KDMS_NORM_MIN` and `KDMS_NORM_MAX`. These two attributes indicate the minimum and maximum magnitude of the data. The effective scale factor and offset are computed by examining every point in the primitive that was accessed via `kdms_get_data()` or `kdms_put_data()`. This is not a global normalization over the entire set, but rather a local normalization over the extent of the data being accessed.

It is important to note the order in which each of these presentation changes is applied. The normalize and scale operations occur after the cast operation if the cast operation is converting to a "higher order" data type, i.e., a data type that has a higher range or precision. If casting from a higher order data type to a lower order data type, one that has less range or precision, then the normalization or scaling occurs before the cast operation.

B.3. Padding and Interpolation

Padding and interpolation are operations that change the apparent size of the data set being accessed. These operations are useful in circumstances where a particular size of data is required in order for an algorithm to function properly, such as a Fast Fourier Transform, or in instances in which two operands must be the same size in order for the algorithm to behave in a predictable manner, such as an addition operation. Other instances where interpolation is useful is in visual applications for zooming or panning windows. This behavior is controlled by an attribute called `KDMS_INTERPOLATE`. This attribute can be set to one of three values: `KNONE`, `KPAD`, or `KINTERPOLATE`. The default value of this attribute is `KPAD`. When this attribute is set to `KNONE`, it indicates that access of data outside of the physical bounds of the data set should not be permitted. If a program attempts to access data that lies beyond the bounds of the data set in this mode, Data Management Services will generate an error.

If set to use the `KPAD` mode, Data Management Services will allow access of data outside of the physical bounds of the data set. Any data that is retrieved that is not part of the data set will be set to a constant value indicated by the `KDMS_PAD_VALUE` attribute. This attribute takes two `double` arguments that represent the real and imaginary component. The imaginary component is only used if the data type being returned is complex.

If the presentation size is set to be larger than the physical size, then any data that falls outside of the bounds of the data set will be similarly set to this pad value. This mode also allows the presentation size to be set to a value that is smaller than the physical size. In this mode, data outside of the presentation size is simply clipped; i.e., it is not accessible.

If the `KDMS_INTERPOLATE` attribute is set to `KZERO_ORDER`, then this indicates that the difference in the presentation size and the physical size of the data segment should be rectified via a zero-order-hold (i.e., pixel replication) interpolation. Currently this is the only true interpolation mode available in Data Services. If the presentation size is larger than the physical size, then an adjacent data point is replicated for each point that does not exist in the interpolated data set. If the presentation size is smaller than the physical size, then the

data set is sub-sampled to produce a smaller version of the original data.

B.4. Conversion of Complex Data

Complex conversion can be thought of as an extension to casting. However, since the process of converting data from a complex data type to a non-complex data type (or visa-versa) is uniquely lossy, this capability is provided as a separate feature so that its behavior can be more easily controlled.

This control is provided via the `KDMS_COMPLEX_CONVERT` attribute (and its sister attributes for each of the other polymorphic segments). This attribute determines how to translate real valued data into complex data. For example, if the `KDMS_COMPLEX_CONVERT` attribute is set to `AccuSoftEAL`, the real valued data will be interpreted as the real part of the complex pair. Similarly, a setting of `KIMAGINARY` instructs data services to interpret the data as the imaginary component. In either case, the other component of the pair is set to zero. When `KDMS_COMPLEX_CONVERT` is set to `KMAGNITUDE`, then the magnitude of the complex pair is set to the value of the data. Currently, this is performed by setting the phase to 0 radians. Thus, `KMAGNITUDE` has the same effect as `AccuSoftEAL`. If the `KDMS_COMPLEX_CONVERT` attribute is set to `KPHASE`, then the real valued data is interpreted as radian data and the magnitude is set to 1.0.

When complex data is returned to the application from Data Services, it will be in the form of a `kcomplex` or `kdcomplex`. There is a complete set of operator functions available for operating on these data types. These functions are available in the *kmath* library. When operating on complex data, the application programmer is encouraged to refer to the *kmath* library for information on complex operations.

B.5. Index Order Manipulation

Every data set has what is referred as an *index order*. Index order describes the ordering of the axes associated with the data in a segment in terms of how it is stored and retrieved. A two dimensional example of this concept is illustrated in Figure 4.

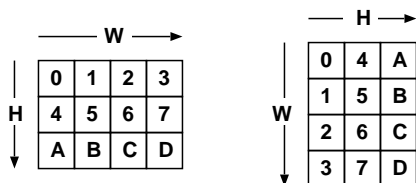


Figure 4: Two index orders associated with the same data. The left data set has an index order that is W-H. The right data set has an H-W index order. Notice that the index order difference between the left and right is effectively a transpose.

In Figure 4, the two data sets are identical--that is, they contain the same data, oriented in an identical fashion across the W and H axes. In other words, given any point, its (W,H) coordinate is the same. Index order merely describes the orientation of data when mapped into a linear address space such as in a data file or in a data array. Figure 5 illustrates the how both data sets would be ordered in an array or file.

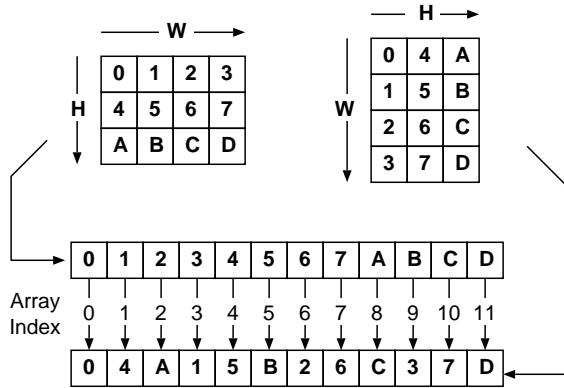


Figure 5: Stored representation of both data sets in Figure 4.

This concept scales to higher dimensions. However, the notion of transposition falls apart. For example, 3 dimensional data oriented along W , H , and D axes can be represented in 6 different orderings: W - H - D , H - W - D , D - H - W , W - D - H , H - D - W , and D - W - H . In fact, the number of unique index orderings N , is a function of the dimensionality d : $N = d!$. Note that in the three dimensional case, most index orders are not transposes of one another. The closest concept that can be used to describe the relationship between two index orders is a reflection of the data across an axis in N space.

Similar to size and data type, index order can be different between the presentation of the data and its physical representation. In such instances, a call to `kdms_get_data()` or `kdms_put_data()` results in a *reflection* or *transpose* as described above. This capability is important because there is no standard orientation in which data is stored in a file that is adhered to by all file formats. For example, even with rasterized RGB color images, some file formats store the red, green and blue components as contiguous vectors, while others will store the red plane as a contiguous band, followed by the green plane and finally by the blue plane. See Figure 6. Regardless of the stored representation of the data, a typical programmer will want to assume a single index ordering so that algorithm development can be emphasized rather than data access.

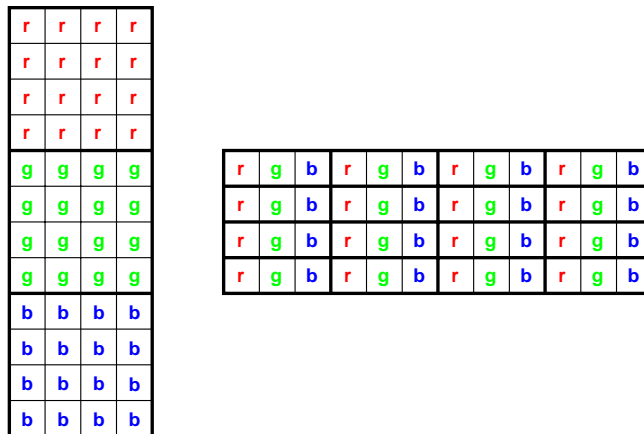


Figure 6: An example of the motivation for the index order capabilities provided by Data Management Services: rasterized image storage. This figure depicts two common storage index orders of a single 4x4 image containing RGB triples for each pixel.

C. Attributes

There are two types of *attributes* in Data Management Services: (1) attributes that affect behavior of a single segment in an object, and (2) attributes that affect the behavior of the whole object. For example, attributes such as data type and size are related to segments, but attributes such as the name of the file containing the data are related to the object as a whole.

Table 1 below contains definitions for all of the predefined global attributes. When using `kdms_get_attribute`, `kdms_set_attribute`, or any other attribute function, the second argument to the function specifies the segment that should be accessed for obtaining the specified attribute. For global attributes, this argument should be set to `KDMS_OBJECT`.

C.1. Global Attributes

Global Attributes		
Attribute and Default	Legal Values	Definition
KDMS_ARCHITECTURE int architecture	KMACH_UNKNOWN KMACH_LITTLE_ENDIAN_IEEE KMACH_LITTLE_ENDIAN_VAX KMACH_LITTLE_ENDIAN_64 KMACH_BIG_ENDIAN_IEEE KMACH_BIG_ENDIAN_CRAY	This attribute is an integer value which encodes a description of the floating point and integer representation for the machine which what used to generate the object. A set of C defines are typically used when operating on the value of this attribute in a program. Typically, this attribute is set based on an examination of the input object, and is set to the local architecture on an output object. The encoding scheme and specific values for these defines can be found in <code>\$BUILD/include/machine/kmachine.h</code> . Persistence: stored
KDMS_COMMENT char * comment NULL		This attribute is a NULL terminated string used to document the object. This attribute is used by a user or programmer to describe the origin or nature of the data set. When this attribute is set, it overwrites anything previously held in this attribute. Therefore, it is up to the programmer to first get the comment attribute, append new information to it, and then set the entire comment attribute, if prior comment information is to be propagated. To clear the comment attribute, pass in NULL when setting the attribute. This attribute is copied with the <code>kdms_copy_object()</code> function calls. Persistence: stored

Global Attributes		
Attribute and Default	Legal Values	Definition
KDMS_COUPLING int coupling See Note 1.	KCOUPLED KDEMAND KUNCOUPLED	<p>When this attribute is set to KCOUPLED, changes to any attribute that affects the physical representation of the data (for example, data type, size, etc.) will be propagated to the physical layer immediately. Otherwise, the presentation layer is the only layer that is changed, --the physical layer remains unchanged. The difference between KUNCOUPLED and KDEMAND is that KDEMAND allows the kdms_sync () function call to force an update of the presentation and physical layers. When this attribute is set to KUNCOUPLED, the calling the kdms_sync () will not do anything. See kdms_sync () for more information.</p> <p>Persistence: transient</p>
KDMS_DATE char * date current date		<p>This attribute is a NULL terminated string used to record the date of the creation of the data object. This attribute is NOT copied by kdms_copy_object (). To assign the current date as defined by computer system, pass in NULL when setting the attribute. The date will be stored in the default format of the UNIX date command ("day month date HH:MM:SS timezone year", e.g. "Wed Mar 10 00:07:23 MST 1994")</p> <p>Persistence: stored</p>
KDMS_FLAGS int flags KNONE	KOBJ_READ KOBJ_WRITE KOBJ_HEADERLESS KOBJ_RAW	<p>This attribute returns the flags used when opening the data object. kdms_input automatically sets KOBJ_READ and kdms_output automatically sets KOBJ_WRITE when called. When used in kdms_open, the legal values can be OR'd together to change the behavior of the object. For example, ORing together KOBJ_READ and KOBJ_RAW together will cause files to be read in as raw data. This attribute is available so that a process can customize its interaction with an object based on how it was opened.</p> <p>Persistence: transient</p>
KDMS_FORMAT char * format viff	kdf viff jpeg pnm pcx xpm xbm xwd eps rast avs ascii raw	<p>This attribute specifies the file format that will be used with the object. If the object is an input object, then this attribute is automatically initialized to the file format that the object is stored in. If the object is an output object, then this attribute defaults to "viff", indicating that the output data file will be a viff. On output objects, this attribute can be set to any of the legal values. The result is that when the object is closed, it will be written out in the format specified.</p>

Global Attributes		
Attribute and Default	Legal Values	Definition
		Persistence: stored
KDMS_FORMAT_DESCRIPTION char * description viff	N/A	This (read-only) attribute retrieves the file format description that will describes the format associated with an object. Persistence: transient
KDMS_NAME char * name		This attribute is used to obtain the filename associated with the specified data object. This is that name passed in to <code>kdms_open</code> , <code>kdms_output</code> , or <code>kdms_input</code> . Objects that are instantiated with <code>kdms_create</code> do not have a filename. In such instances, this attribute's value is NULL. Persistence: stored
KDMS_RAW_OFFSET int offset 0		This attribute specifies an offset to use when opening raw data files for reading. The file pointer will be moved to the offset specified before the reading begins. The offset is in bytes. Persistence: transient

Table 2 below contains definitions for all of the predefined segment attributes.

C.2. Segment Attributes

Segment Attributes		
Attribute and Default	Legal Values	Definition
KDMS_BUFFER_THRESHOLD int threshold 2097152 bytes	> 0	This attribute is used to specify the largest number of bytes that should be allocated for buffering data in memory. This number is used to determine the size and geometry of a buffer that is used to minimize transport access. Thus, it has a large impact on the performance of a process using to determine the size and geometry of the <code>KDMS_OPTIMAL_REGION_SIZE</code> . Persistence: transient

Segment Attributes		
Attribute and Default	Legal Values	Definition
KDMS_COMPLEX_CONVERT int convert KREAL	KIMAGINARY KMAGNITUDE KPHASE KREAL KMAGSQ KMAGSQP1 KLOGMAG KLOGMAGP1	<p>This attribute specifies how complex data should be converted. If it is converted to a "lower" data type, this attribute specifies how to down-convert the data. For example if the data is actually complex, but the presentation attribute is byte, the complex data would first be converted to the representation defined by this attribute, and then converted to byte.</p> <p>If the data is being converted from a "lower" data type to a complex data type, this attribute defines how the data should be interpreted — as the real or imaginary component of the complex pair. KPHASE and KMAGNITUDE are invalid values for up converting to complex, and will result in an error.</p> <p>Persistence: transient</p>
KDMS_DATA_TYPE int datatype	KBIT KBYTE KUBYTE KSHORT KUSHORT KINT KUINT KLONG KULONG KFLOAT KDOUBLE KCOMPLEX KDCOMPLEX	<p>This attribute is used to get or set the data type, or numerical representation of the data. This data type will be the presentation data type, not necessarily the physical data type. See the KDMS_COUPLING attribute for more information on how to control the presentation and physical data types. When the application programmer specifies a presentation data type that is different than the actual data type of the stored data, the <code>kdms_get_data</code> function will convert the data to return the requested data type. Likewise, the <code>kdms_put_data</code> function expects data that is in the data type specified by this attribute to the output object, and if the data being "put" is of a different type, it will be converted. This attribute must be set for objects created via <code>kdms_create</code> or output objects that are opened with <code>kpds_output</code> or <code>kpds_open</code>, or else the get and put data calls will fail.</p> <p>Persistence: stored</p>
KDMS_DIMENSION int dim		<p>This attribute is used to get or set the dimensionality of a data segment. For example, a signal is a 1-D data set, so a special segment to store a single signal could have a dimensionality of 1.</p> <p>Persistence: stored</p>

Segment Attributes		
Attribute and Default	Legal Values	Definition
KDMS_INDEX_ORDER int * order		<p>This attribute specifies the ordering of indices of the segment to set the orientation of how data is stored and retrieved. When the application programmer specifies a presentation index order different than the actual index order of the stored data by setting this attribute, the get functions will acquire the data using the presentation index order. This attribute also dictates the interpretation of the KDMS_SIZE and KDMS_OPTIMAL_REGION_SIZE attributes. For example, if the index order is KWIDTH, KDEPTH, KHEIGHT on a 3 dimensional segment, then in the size array used when setting or getting the KDMS_SIZE attribute, the arguments will be width, depth, and height, in that order.</p> <p>Persistence: stored</p>
KDMS_INTERPOLATE int interpolate KPAD	KNONE KPAD KZERO_ORDER	<p>This attribute specifies how the data should be presented if the application program requests a size different from what is physically stored. If the size requested is larger than the physical size and the interpolation requested is KPAD the pad value will be returned for all points outside of the physical size. If the size requested is smaller than the physical size and the interpolation requested is KPAD the returned data is clipped to the size requested. If the size requested is larger than the physical size and the interpolation requested is KZERO_ORDER the data is duplicated. If the size requested is smaller than the physical size and the interpolation requested is KZERO_ORDER the data is sub-sampled. If the interpolate attribute is set to KNONE, an error will be returned if the program requests a size different from what is physically stored.</p> <p>Persistence: transient</p>
KDMS_NORM_MAX double norm_max	> norm_min	<p>This attribute specifies the maximum to be used when normalizing data values. This attribute is used in conjunction with the KDMS_NORM_MIN attribute, respectively, to determine the bounds of a normalization operation. This attribute comes into play when the KDMS_SCALING attribute is set to KNORMALIZE.</p> <p>Persistence: transient</p>
KDMS_NORM_MIN double norm_min unknown	< norm_max	<p>This attribute specifies the minimum to be used when normalizing data values. This attribute is used in conjunction with the KDMS_NORM_MAX attribute, respectively, to determine the bounds of a normalization operation. This attribute comes into play when the KDMS_SCALING attribute is set to KNORMALIZE.</p> <p>Persistence: transient</p>

Segment Attributes		
Attribute and Default	Legal Values	Definition
KDMS_OFFSET int * offset 0, 0, 0, 0, 0		This attribute specifies a relative offset from zero of the origin of the data set. This value will be added to the begin and end corner markers on all get and put data operations. Persistence: transient
KDMS_OPTIMAL_REGION_SIZE int * sizes	> 0	This attribute will return the size of a region of data and the number of such regions that is most efficient to process in terms of performance and memory use. The number of values in the integer array is n+1, where n is the dimension of the segment. The first n entries correspond to the axes of the data set and the last entry in the array is the number such regions in the segment. Persistence: transient
KDMS_PAD_VALUE double real 0.0 imag 0.0		This attribute specifies the real (and imaginary) values of the pad data if the KDMS_INTERPOLATE attribute is set to KPAD, respectively. The double values must be specified, whether the data is real or complex. The pad values will internally be converted from double to the appropriate data type. Persistence: transient
KDMS_PHYSICAL_DATA_TYPE int dim	KBIT KBYTE KUBYTE KSHORT KUSHORT KINT KUINT KLONG KULONG KFLOAT KDOUBLE KCOMPLEX KDCOMPLEX	This attribute is used to get the data type, or numerical representation of the data at the physical level. Ordinarily, attribute setting and getting retrieve the presentation value of an attribute. However, there are some rare instances where direct access to the physical representation of the data is necessary, but it is desirable to maintain the existing presentation attributes. Otherwise, this attribute is identical to KDMS_DATA_TYPE. It is the difference between the value of this attribute and the value of the KDMS_DATA_TYPE attribute that will cause a cast operation to occur during calls to kpds_get_data() and kpds_put_data(). Persistence: stored

Segment Attributes		
Attribute and Default	Legal Values	Definition
KDMS_PHYSICAL_DIMENSION int dim	> 0	<p>This attribute is used to get the dimensionality of the data at the physical level. Ordinarily, attribute setting and getting retrieve the presentation value of an attribute. However, there are some rare instances where direct access to the physical representation of the data is necessary, but it is desirable to maintain the existing presentation attributes. Otherwise, this attribute is identical to KDMS_DIMENSION. Currently, there should be no difference between this value and the value of KDMS_DIMENSION.</p> <p>Persistence: stored</p>
KDMS_PHYSICAL_INDEX_ORDER int * order		<p>This attribute is used to get the index order of the data at the physical level. Ordinarily, attribute setting and getting retrieve the presentation value of an attribute. However, there are some rare instances where direct access to the physical representation of the data is necessary, but it is desirable to maintain the existing presentation attributes. Otherwise, this attribute is identical to KDMS_INDEX_ORDER. It is the difference between the value of this attribute and the value of the KDMS_INDEX_ORDER attribute that can cause a re-ordering of the data, similar to a matrix transposition during calls to kpds_get_data() and kpds_put_data().</p> <p>Persistence: stored</p>
KDMS_PHYSICAL_SIZE int * size		<p>This attribute is used to get the size of the data at the physical level. Ordinarily, attribute setting and getting retrieve the presentation value of an attribute. However, there are some rare instances where direct access to the physical representation of the data is necessary, but it is desirable to maintain the existing presentation attributes. Otherwise, this attribute is identical to KDMS_SIZE. It is the difference between the value of this attribute and the value of the KDMS_SIZE attribute that can cause an interpolation or padding to occur during calls to kpds_get_data() and kpds_put_data() based on the value of the KDMS_INTERPOLATE attribute.</p> <p>Persistence: stored</p>
KDMS_SCALE_FACTOR double scale_factor 1.0		<p>This attribute specifies the scaling factor to be used when scaling data values. This attribute comes into play when the KDMS_SCALING attribute is set to KSCALE, respectively.</p> <p>Persistence: transient</p>

Segment Attributes		
Attribute and Default	Legal Values	Definition
KDMS_SCALE_OFFSET double real 0.0 offset 0.0		This attribute specifies the scaling offset to be used when scaling data values. This attribute comes into play when the KDMS_SCALING attribute is set to KSCALE, respectively. Persistence: transient
KDMS_SCALING int scaling KNONE	KNONE KNORMALIZE KSCALE	This attribute specifies whether scaling or normalization should be performed. If KSCALE is specified for a segment, data values from that segment will be scaled, according to the KDMS_SCALE_FACTOR and KDMS_SCALE_OFFSET attributes. If KNORMALIZE is specified for a segment, data values from that segment will be normalized using the KDMS_NORM_MIN and KDMS_NORM_MAX attributes. If this attribute is set to KNONE for the a segment, data values from that segment will not be scaled or normalized. Persistence: transient
KDMS_SIZE int * size	> 0	This attribute specifies the size of the axes of a data segment. When the application programmer specifies a size larger than the actual size of stored data, the get functions will sub-sampled, clipped, padded or duplicated the data to present the program with the requested amount, see the attribute KDMS_INTERPOLATE for more details. The put functions store exactly the size that the physical attributes will allow even if the amount of data "put" (set by the presentation attributes) is different. This attribute must be set for objects created via kpds_create_object or output objects else get/put data calls will fail. Persistence: stored
KDMS_TRANSFERABLE int dim TRUE	TRUE FALSE	This attribute is used to determine if a segment is copied to a destination via kdms_copy_object(). If set to then kdms_copy_object() will copy the segment to the destination object. If it is set to FALSE, then it will not be copied with the other segments in the data set. This capability is used to some applications to store information which will be invalid if other data segments are operated on, such as statistical information. Persistence: stored

D. Functions Provided By Data Management Services

The multiple attribute functions `kpds_get_attributes()`, `kpds_match_attributes()`, and `kpds_set_attributes()` require a NULL at the end of the variable argument list to indicate the end of the list.

D.1. Object Management

- `kdms_create()` - create a temporary data object.
- `kdms_open()` - create an object associated with an input or output transport.
- `kdms_close()` - close an open data object.
- `kdms_reopen()` - associate new data with an existing object
- `kdms_reference()` - create a reference of a data object.
- `kdms_sync()` - synchronize physical and presentation layers of a data object.
- `kdms_update_references()` - update segment presentation of all reference objects.
- `kdms_close_hook()` - insert a service to be called when an object is closed.
- `kdms_reference_list()` - return a klist of references.
- `kdms_get_segment_names()` - get an array of segment names for the object specified.

D.1.1. `kdms_create()` — *create a temporary data object.*

Synopsis

```
kobject
kdms_create(void)
```

Returns

kobject on success, KOBJECT_INVALID upon failure

Description

`kdms_create` is used to instantiate a data object (`kobject`) when it will only be used for temporary storage of information. If you are intending to process an object that already exists as a file or transport (input), or you are planning on saving the `kobject` to a file or transport (output), then the appropriate routines to use are `kdms_input`, `kdms_output`, or `kdms_open`.

This function creates an instance of a data object that will have associated with it a temporary transport that will be used for buffering large amounts of data. This temporary transport will be automatically removed when the process terminates. There is no way to rename the temporary file or replace the temporary file with a permanent one.

The `kdms_create` function call creates what is essentially a "blank" object. That is, the object will initially have no segments, and almost all global attributes will be initialized to default values. If a default is not appropriate, then the attribute will be uninitialized. The default values for attributes are described in Chapter 5 of Program Services Volume 2. of the Khoros 2.0 Manual.

An object that is created with this function call behaves similarly to an output object that is created with the `kdms_output` function call. Thus, it is necessary to create each segment that is desired and

initialize attributes such as size and datatype prior to using the object.

D.1.2. kdms_open() — *create an object associated with an input or output transport.*

Synopsis

```
kobject
kdms_open(
    char *name,
    int  flags)
```

Input Arguments

name

a string that contains the path name of a file or transport that will be associated with the object.

flags

how the object is to be opened. a combination of KOBJ_READ, KOBJ_WRITE, KOBJ_RAW as described above.

Returns

kobject on success, KOBJECT_INVALID upon failure

Description

kdms_open is used to instantiate a data object (kobject) that is associated with a permanent file or transport. If a permanent file is not desired (i.e. the object is going to be used as temporary storage, and will not be used by any other process) then the kdms_create function call should be used instead.

The first argument to this function is the transport or file name. This argument indicates the name of the transport that is associated with the object. The transport name can be any legal khoros transport description. While this will usually be a regular UNIX file name, it is also possible to specify such things as shared memory pointers, memory map pointers, sockets, streams, and even transports on other machines. For more information on the syntax of a VisiQuest transport name, refer to the online man page for the VisiQuest function kopen.

The second argument to the kdms_open function call is used to provide data services with specific information about how the object is going to be manipulated. The flags argument is analogous to kopen's flags argument. The flags argument is constructed by bitwise OR'ing predefined values from the following list:

KOBJ_READ

Open an existing file or transport for reading (input). By using this flag, you are indicating that the file or transport exists and that it contains valid data. If it does not exist, or the data is not recognized, then an error message will be generated

and this function will return `KOBJECT_INVALID`.

`KOBJ_WRITE`

Open a file or transport for writing (output). By using this flag, you are indicating that any data that you write to the object will be stored in the file or transport specified.

`KOBJ_RAW`

When an object is opened, data services usually attempts to recognize the file format by examining the first part of the file. By setting this value, you will bypass this operation, forcing the file to be read as raw unformatted data.

These flags can be combined arbitrarily (with the exceptions given above) to alter the interpretation of the file or transport. For example, specifying both `KOBJ_READ` and `KOBJ_WRITE` will result in a read/write file object. This implies that the file already exists and will be read from using `kdms_get_data` and written to using `kdms_put_data`. When `kdms_close` is called, the changes that are a result of calls to `kdms_put_data` will be stored to the file.

However, if you intend to open an output object, but you need to occasionally read data from it that you have already written, it is not necessary to specify `KOBJ_READ` (in fact, doing so may result result in an error if the file or transport does not already exist).

Likewise, it is possible to call `kdms_put_data` on an input object (one which was opened without the `KOBJ_WRITE` flag). If this is done, then subsequent calls to `kdms_get_data` on a region that has been written to will contain the new data. However, the file that is associated with this input object will not be changed. Thus, the `KOBJ_READ` and `KOBJ_WRITE` flags only indicate what operations are allowed on the permanent file that is associated with the object, not what operations are allowable on the object itself.

If `KOBJ_READ` is specified, then the Data Services will attempt to recognize the file format automatically. If it fails, then this function will return `KOBJECT_INVALID`, indicating that it was unable to open the object, unless the `KOBJ_RAW` flag was also specified, in which case, it will assume that the input file is simply headerless data. The structured file formats that are currently recognized are Kdf (The Khoros 2.0 standard file format), Viff (The Khoros 1.0 standard file format, Pnm (Portable Any Map, which includes PBM, PGM, and PNM), and Sun Raster.

Restrictions

The `KOBJ_RAW` flag will have unpredictable results if it is combined with the `KOBJ_WRITE` flag. This limitation will be removed in a later release of the Khoros 2.0 system.

The Kdbm file format has a bug in it that prevents it from being used on a stream base output. This will be fixed in a future release of the Khoros 2.0 system.

D.1.3. `kdms_close()` — *close an open data object.*

Synopsis

```
int
kdms_close(kobject object)
```

Input Arguments

`object`
the object to be closed.

Returns

TRUE on success, FALSE otherwise

Description

This function is called on an object when all interaction with the object is completed. In addition to freeing resources that were used to manage the object, this function also, performs any "last minute" manipulation on the file or transport that is associated with the object.

If the object was created with the `kdms_reference` function call, or if another object was created as a reference of the one being closed, then the object might be sharing some of its resources with other objects. If this is the case, then those shared resources will not be freed, but rather they will be disassociated from the object being closed. Thus, closing an object does not affect any other object.

D.1.4. `kdms_reopen()` — *associate new data with an existing object*

Synopsis

```
kobject
kdms_reopen(
    kobject object,
    char *name,
    int flags)
```

Input Arguments

`object`
data object re-open filename - transport name of new data set to associate with the object.
`flags`
flags to use when opening the data set specified by the filename argument.

Returns

object (the input argument) on success, NULL on failure.

Description

This function is used to associate new stored data with an already open data object. This operation can only be performed on an input object. This function is typically used in an interactive environment when many references to replace the data set being operated on or visualized with a new data set, without having to close all of the references and replace them with new ones.

D.1.5. `kdms_reference()` — *create a reference of a data object.*

Synopsis

```
kobject  
kdms_reference(kobject object)
```

Input Arguments

object
the abstract data object to be reference.

Returns

a kobject that is a reference of the input object on success, KOBJECT_INVALID upon failure

Description

This function is used to create a reference of a data object that can be treated as a second independent data object under most circumstances. A referenced object is similar conceptually to a symbolic link in a UNIX file system in most respects. For example, getting data from an input object and a reference of the object will result in the same data. Data that is put on an output object can then be gotten from one of its references.

The similarity ends there. Once an object is referenced, the two resulting objects are equivalent--there is no way to distinguish the original from the reference. In fact, closing the original does not in any way affect the reference, and visa-versa.

`kdms_reference` creates a new object that has presentation attributes that are independent of the original object's presentation attributes. The presentation attributes are UNCOUPLED from the physical attributes, see the description found in Chapter 6 of the the VisiQuest Programmer's Manual on the KDMS_COUPLING attribute for more information. The two objects (or more if there are future calls to `kdms_reference`) share all physical resources.

D.1.6. kdms_sync() — *synchronize physical and presentation layers of a data object.*

Synopsis

```
int
kdms_sync(
    kobject object,
    char *segment,
    int direction)
```

Input Arguments

object

data object to be resynchronized.

segment

segment re-synchronize. If this is set to KDMS_OBJECT, then all segments are re-synchronized.

direction

the desired direction of the synchronization. the legal values are KPRES2PHYS, which indicates that the physical layer will be updated to correspond to the presentation layer; and KPHYS2PRES, which indicates that the presentation layer will be updated to correspond to the physical layer.

Returns

TRUE on success, FALSE otherwise

Description

This function is used to update physical attributes of a data object to match those of the presentation layer, or visa-versa. When an attribute is set via `kdms_set_attribute(s)` or `kdms_copy_attribute(s)` calls, the presentation version of the attribute is the only thing that is directly manipulated. The `KDMS_COUPLING` attribute is used at that time to determine if the physical attribute should be updated to correspond to its value at the presentation level. The `KDMS_COUPLING` attribute can take on one of three values: `KUNCOUPLED`, `KCOUPLED`, or `KDEMAND`. If it is set to `KUNCOUPLED` or `KDEMAND`, then Data Services will not update the physical layer. If the attribute is set to `KCOUPLED`, then data services immediately updates the physical layer. If the attribute is set to `KDEMAND`, then this updating will only occur when `kdms_sync` is called. If the `KDMS_COUPLING` attribute is set to `KUNCOUPLED`, then this routine will simply return, without issuing an error message.

D.1.7. kdms_update_references() — *update segment presentation of all reference objects.*

Synopsis

```
int
kdms_update_references(
```

```
kobject object,  
char *segment)
```

Input Arguments

object
object containing segment to propagate.
segment
name of segment to be propagated.

Returns

TRUE on success, FALSE otherwise

Description

This function propagates the values of the presentation attributes from the specified object and segment to the corresponding segments in all of the object's references.

D.1.8. kdms_close_hook() — <i>insert a service to be called when an object is closed.</i>
--

Synopsis

```
kfunc_int  
kdms_close_hook(  
    kobject object,  
    int (*func) PROTO((kobject)))
```

Input Arguments

object
object to have close hook function added to.
func
function to set as close hook. This function has the following prototype: func(kobject).

Returns

The close_hook that was assigned prior to this new assignment.

Description

This function is used to insert a special function that is called immediately before an object is closed (after a call to kdms_close) that can perform any cleanup that may be required before the object is written if it is an output object.

D.1.9. `kdms_reference_list()` — *return a klist of references.*

Synopsis

```
klist *  
kdms_reference_list(kobject object)
```

Input Arguments

`object`
the object to get references of.

Returns

a `klist *` on success, `NULL` on failure.

Description

This function returns a `klist` of objects that are references of the object passed in as the argument to this function. The object passed in will also be in this list. NOTE: This list is the one actually used by KDMS to manage references. Destroying this list will cause grief like you've never seen before.

D.1.10. `kdms_get_segment_names()` — *get an array of segment names for the object specified.*

Synopsis

```
char **  
kdms_get_segment_names(kobject object, int *number)
```

Input Arguments

`object`
the object to get the segment names from.

Output Arguments

`number`
the number of segments in the object (and thus, the number strings in the array that is returned).

Returns

an array of strings containing the names of the segments present in the object on success, `NULL` on failure.

Description

Given an object, obtain an array of strings which are the names of all segments which exist in the object.

D.2. Information

- *kdms_support()* - obtain a list of file formats supported by data services.

D.2.1. *kdms_support()* — *obtain a list of file formats supported by data services.*

Synopsis

```
char **  
kdms_support (  
    int *num,  
    int list_inputs,  
    int list_outputs)
```

Input Arguments

```
list_inputs  
    TRUE if formats with input support should be included in the list  
list_outputs  
    TRUE if formats with output support should be included in the list
```

Output Arguments

```
num  
    The number of formats in the returned string array.
```

Returns

An array of strings (char **) containing a list of all formats that are currently defined in Data Services.

Description

This function is used to obtain a list of the file formats supported by the data abstraction.

D.3. Segment Management

- *kdms_query_segment()* - determine if a data segment is available.
- *kdms_create_segment()* - create a segment on a data object.
- *kdms_destroy_segment()* - destroy a segment from a data object.
- *kdms_rename_segment()* - rename a segment

D.3.1. `kdms_query_segment()` — *determine if a data segment is available.*

Synopsis

```
int
kdms_query_segment (
    kobject object,
    char *segment)
```

Input Arguments

`object`
The object that segment may exist in.

`segment`
The name of segment to determine the existence of.

Returns

TRUE if the data segment exists, FALSE otherwise

Description

This function is used to determine if a data segment in an object is available. It returns TRUE if the data segment exists, FALSE otherwise.

D.3.2. `kdms_create_segment()` — *create a segment on a data object.*

Synopsis

```
int
kdms_create_segment (
    kobject object,
    char *segment)
```

Input Arguments

`object`
The object to create the new segment in.

`segment`
The name of segment to create.

Returns

TRUE on success, FALSE otherwise

Description

This function is used to create a segment that does not already exist in a data object. If the segment already exists, then this function will generate an error.

D.3.3. `kdms_destroy_segment()` — *destroy a segment from a data object.*

Synopsis

```
int
kdms_destroy_segment(
    kobject object,
    char *segment)
```

Input Arguments

`object`
The object containing the segment to be destroyed.

`segment`
The name of segment to be destroyed.

Returns

TRUE on success, FALSE otherwise

Description

This function is used to destroy an existing segment from a data object. Once a segment has been destroyed, any data or attributes associated with that segment will be lost forever. A new segment can be created in its place.

If the segment does not exist, then an error message will be issued.

D.3.4. `kdms_rename_segment()` — *rename a segment*

Synopsis

```
int
kdms_rename_segment(
    kobject object,
    char *oname,
    char *nname)
```

Input Arguments

object

The object to rename a segment on. old_name - The current name of the segment to be renamed.
new_name - The new name of the segment.

Returns

TRUE if successful, FALSE otherwise

Description

This function is used to rename a segment in an open object.

D.4. Attribute Management

- *kdms_define_quasi_attribute()* - define a quasi attribute
- *kdms_define_attribute()* - define an attribute for for a session
- *kdms_undefine_attribute()* - undefine a defined attribute
- *kdms_query_attribute_definition()* - determines if an attribute is defined.
- *kdms_create_attribute()* - instantiate an attribute
- *kdms_destroy_attribute()* - destroy an attribute
- *kdms_vset_attribute()* - open varargs set attribute
- *kdms_vset_attributes()* - set attributes on a kvalist
- *kdms_set_attribute()* - set the value of an attribute
- *kdms_set_attributes()* - sets the values of multiple attributes
- *kdms_vget_attribute()* - get a single attribute on a kvalist
- *kdms_vget_attributes()* - get attributes on a kvalist
- *kdms_get_attribute()* - get the value of an attribute within a segment of an abstract object.
- *kdms_get_attributes()* - gets the values of a variable number of attributes within a single segment of an object.
- *kdms_match_attribute()* - returns TRUE if the same segment attribute in two abstract data objects match.
- *kdms_vmatch_attributes()* - returns true if the vararg list of segment attributes in two abstract data objects match.
- *kdms_match_attributes()* - returns true if the list of segment attributes in two abstract data objects match.
- *kdms_copy_attribute()* - copy an attribute from a source object to a destination object.
- *kdms_vcopy_attributes()* - copy attributes given in a kvalist
- *kdms_copy_attributes()* - copy attributes from a source object to a destination object.
- *kdms_query_attribute()* - get information about an attribute
- *kdms_print_attribute()* - print the value of an attribute
- *kdms_get_attribute_names()* - get a list of attributes from an object.

D.4.1. *kdms_define_quasi_attribute()* — *define a quasi attribute*

Synopsis

```
int kdms_define_quasi_attribute(
```

```

char *association,
char *attribute,
kaddr client_data,
kfunc_int *get,
kfunc_int *set,
kfunc_int *match,
kfunc_int *copy,
kfunc_int *query,
kfunc_int *print)

```

Input Arguments

`association`

string indicating where it is legal to to invoke the attribute. NULL implies the attribute can be invoked at the object level. A segment name implies that the attribute can only be invoked on that segment. The identifier KDMS_ALL_SEGMENTS implies that the attribute can be invoked for any segment, but not for the object.

`attribute`

attribute string identifier.

`client_data`

pointer to client data. This client data will be passed in to all the the routines for this attribute.

`get`

get handler function for this attribute. This function will be invoked whenever a `kdms_get_attribute`, `kdms_get_attributes`, `kdms_vget_attribute`, or `kdms_vget_attributes` function is called and the given segment matches the definition's association.

The get handler declaration is of the form :

```

int get_handler(
    kobject object,
    ktoken association,
    ktoken attribute,
    kaddr client_data,
    kva_list *list)

```

The object is passed from the calling get function. The association is the tokenized representation of the segment argument from the get function. The attribute is the tokenized representation of the attribute name. The client_data is the client_data from this definition. The variable argument list is an already opened varargs list which contains the pointers with which to return the attribute data. The variable arguments can be pulled off with the `kva_arg()` function. The list will be closed after this handler returns. The return value of this handler will also be propagated up to be the return value of the calling get function.

`set`

set handler function for this attribute. This function will be invoked whenever a `kdms_set_attribute`,

kdms_set_attributes, kdms_vset_attribute, or kdms_vset_attributes function is called and the given segment matches the definition's association.

The set handler declaration is of the form :

```
int set_handler(  
    kobject object,  
    ktoken association,  
    ktoken attribute,  
    kaddr client_data,  
    kva_list *list)
```

The object is passed from the calling set function. The association is the tokenized representation of the segment argument from the set function. The attribute is the tokenized representation of the attribute name. The client_data is the client_data from this definition. The variable argument list is an already opened varargs list which contains the data with which to set the attribute data. The variable arguments can be pulled off with the kva_arg() function. The list will be closed after this handler returns. The return value of this handler will also be propagated up to be the return value of the calling set function.

match

match handler function for this attribute. This function will be invoked whenever a kdms_match_attribute, kdms_match_attributes, or kdms_vmatch_attributes function is called and the given segment matches the definition's association.

The match handler declaration is of the form :

```
int match_handler(  
    kobject object1,  
    kobject object2,  
    ktoken association,  
    ktoken attribute,  
    kaddr client_data1,  
    kaddr client_data2)
```

The object1 and object2 arguments are passed from the calling match function. The association is the tokenized representation of the segment argument from the match function. The attribute is the tokenized representation of the attribute name. The client_data1 and client_data2 arguments are from from this definition. The return value of this handler will also be propagated up to be the return value of the calling match function.

copy

copy handler function for this attribute. This function will be invoked whenever a kdms_copy_attribute, kdms_copy_attributes, or kdms_vcopy_attributes function is called and the given segment matches the definition's association.

The copy handler declaration is of the form :

```
int copy_handler(  
    kobject object1,  
    kobject object2,  
    ktoken association,  
    ktoken attribute,  
    kaddr client_data1,  
    kaddr client_data2)
```

The object1 and object2 arguments are passed from the calling copy function. The association is the tokenized representation of the segment argument from the copy function. The attribute is the tokenized representation of the attribute name. The client_data1 and client_data2 arguments are from from this definition. The return value of this handler will also be propagated up to be the return value of the calling copy function.

`query`

query handler function for this attribute. This function will be invoked whenever a `kdms_query_attribute` function is called and the given segment matches the definition's association.

The query handler declaration is of the form :

```
int query_handler(  
    kobject object,  
    ktoken association,  
    ktoken attribute,  
    kaddr client_data,  
    int *num_args,  
    int *arg_size,  
    int *data_type,  
    int *permanent)
```

The object is passed in from the calling query function. The association is the tokenized representation of the segment argument from the query function. The attribute is the tokenized representation of the attribute name. The client_data argument is from from this definition. The num_args argument can be used to return the expected number of arguments for this attribute. The arg_size argument can be used to return the expected argument size for this attribute. The data_type argument can be used to return the expected data type for this attribute. The permanent argument can be used to indicate if this attribute is stored or not. Note that it is up to the programmer to ensure that the values which the handler returns for any of those arguments matches what is going to be processed in the get and set handlers. The return value of this handler will be propagated up to be the return value of the calling query function. In general, a value of TRUE is interpreted to mean that the attribute 'exists'.

`print`

print handler function for this attribute. This function will be invoked whenever a `kdms_print_attribute` function is called and the given segment matches the definition's association.

The print handler declaration is of the form :

```
int print_handler(  
    kobject object,  
    ktoken association,  
    ktoken attribute,  
    kaddr client_data,  
    kfile *outfile)
```

The object is passed in from the calling print function. The association is the tokenized representation of the segment argument from the print function. The attribute is the tokenized representation of the attribute name. The client_data argument is from from this definition. The outfile argument is the (hopefully) open khoros transort which this handler can use to print the attribute to. The return value of this handler will be propogated up to be the return value of the calling print function.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine will define a data services quasi-attribute for a session. A quasi-attribute is one which does not technically have any data associated with it, but one for which a given action will be invoked. From the higher level programmer's point of view, a quasi-attribute will behave identically to a generic attribute. The higher-level function calls will be identical. That is, the same get, set, match, query, and print attribute functions will work on both quasi-attributes and generic attributes. Quasi-attributes are provided simply to allow an application service programmer the ability to provide their own custom functionality for an attribute.

This quasi-attribute definition will be used to determine what action should be taken on a get, set, match, copy or print attribute call. On any kdms attribute call for this attribute, the relevant action handler provided in this definition will be called or invoked.

This attribute definition is distinguished by the attribute name provided here. The name must be a string, unique for the given association.

In each attribute call, a segment argument will be provided to give scope to the attribute. The association given in this definition is used to determine the allowable scope of the attribute. Attributes can be either scoped to the main object (object-level) or scoped to any of the object's data segments (segment-level).

The provided association is simply a string. A NULL association indicates that the attribute can only be invoked when NULL is passed in as the segment argument. An association of KDMS_ALL_SEGMENTS indicates that the attribute can be invoked at the segment level for any segment name except for NULL. If a specific segment name, such as "value" is given for the association, then the attribute can only invoked when "value" is provided as the segment name.

This quasi-attribute can be undefined with the kdms_undefine_attribute call.

D.4.2. `kdms_define_attribute()` — *define an attribute for for a session*

Synopsis

```
int kdms_define_attribute(  
  
    char *association,  
    char *attribute,  
    int  numargs,  
    int  argsize,  
    int  datatype,  
    int  permanent,  
    int  shared,  
    kvalist)
```

Input Arguments

`association`

string indicating where it is legal to to create the attribute. NULL implies the attribute can be instantiated at the object level. A segment name implies that the attribute can only be established on that segment. The identifier `KDMS_ALL_SEGMENTS` implies that the attribute can be established on any segment, but not on the object.

`attribute`

attribute string identifier

`numargs`

number of arguments in the attribute; must be > 0;

`argsize`

number of units of the data type for each attribute argument must be > 0;

`datatype`

data type of the attribute can be `KBIT`, `KUBYTE`, `KBYTE`, `KUSHORT`, `KSHORT`, `KUINT`, `KINT`, `KULONG`, `KLONG`, `KFLOAT`, `KDOUBLE`, `KCOMPLEX`, `KDCOMPLEX`, or `KSTRING`

`permanent`

TRUE if attribute is permanent

`shared`

TRUE if attribute is shared

`kvalist`

the default value in a variable argument list containing data in the form :

value1 [, value2, ...]

The list should consist of "numarg" arguments, where each argument is of "argsize" size of "datatype"

data type.

Returns

TRUE on success, FALSE otherwise

Description

This routine will define a data services attribute for a session. This attribute definition will be used as a template for creating instances of the attribute on an actual data object. This creation will be implicit when the attribute is accessed by any other kdms attribute routine.

The attribute definition is distinguished by the given attribute name. The name must be a string, unique for the given association. The attribute acts as storage for multiple argument values. Each attribute argument can be of any size (> 0) or data type. The number of arguments, the argument size, and the data type are all specified in this definition.

Attributes can be permanent, implying that they are stored to the disk when the object is closed. Technically, a permanent attribute will only be stored if the underlying file format has the capacity for storing it. See the man page for kdatafmt for more information of file formats.

Attributes can be either created on the main object (object-level) or created on any of the object's data segments (segment-level). Whether an attribute is created at the object-level or at the segment-level is determined by the attribute definition's association.

The provided association is simply a string. A NULL association indicates that the attribute can only be created at the object-level. An association of KDMS_ALL_SEGMENTS indicates that the attribute can be created at the segment level on any segment name. If a specific segment name, such as "value" is given for the association, then the attribute can only be created on the "value" segment.

The attribute can be either permanent or transient. Permanent implies that when the object is closed, a representation of the attribute will be written out to the disk. Note that this representation completely describes the attribute, and it will be possible to read this attribute later in another session even if it has not been defined there.

In addition to being created either at the object or segment level, an attribute can also be created at the physical or presentation level.

Changes to physical level attributes are visible to all references of the data object and are termed as "shared" attributes. Changes to presentation level attributes are visible only to the reference object on which the change was made, and thus the attribute is termed as "unshared". Since all references may have potentially different values, unshared attributes may not be permanent.

A default must be provided. This default must consist of the proper number of arguments where each argument is of the same size and data type specified in the definition. This default will be used when initializing any attributes created from this definition.

This attribute can be undefined with the kdms_undefine_attribute call.

D.4.3. `kdms_undefine_attribute()` — *undefine a defined attribute*

Synopsis

```
int kdms_undefine_attribute(  
  
    char *association,  
    char *attribute)
```

Input Arguments

`association`
string indicating the scope of the attribute. The same association that was used to define the attribute must be used here.

`attribute`
attribute string identifier

Returns

TRUE on success, FALSE otherwise

Description

This function will remove an attribute definition from the definition list. The attribute definition corresponding to the given association and attribute name will be removed. This has no effect on any attribute which may have already been instantiated on an object. This function can be used to undefine both generic and quasi attributes.

D.4.4. `kdms_query_attribute_definition()` — *determines if an attribute is defined.*

Synopsis

```
int kdms_query_attribute_definition(  
  
    char *association,  
    char *attribute)
```

Input Arguments

`association`
string indicating the scope of the attribute. The same association that was used to define the attribute must be used here.

attribute
attribute string identifier

Returns

TRUE if attribute is defined, FALSE otherwise

Description

This function will check to see if an attribute is defined on the definition list. The attribute definition corresponding to the given association and attribute name will be searched for.

D.4.5. `kdms_create_attribute()` — *instantiate an attribute*

Synopsis

```
int kdms_create_attribute(  
  
    kobject  object,  
    char     *segment,  
    char     *attribute,  
    int      numargs,  
    int      argsize,  
    int      datatype,  
    int      permanent,  
    int      shared)
```

Input Arguments

object
the object on which to instantiate the new attribute.

segment
segment identifier string specifying which segment to create the attribute in. If NULL, then the attribute will be created at the object level.

attribute
attribute identifier string. This identifier must be unique for the given segment.

numargs
number of arguments in the attribute must be > 0;

argsize
number of units of the data type for each attribute argument must be > 0;

datatype

data type of the attribute can be KBIT, KUBYTE, KBYTE, KUSHORT, KSHORT, KUINT, KINT, KULONG, KLONG, KFLOAT, KDOUBLE, KCOMPLEX, KDCOMPLEX, or KSTRING

permanent

TRUE if attribute is permanent.

shared

TRUE if attribute is shared

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine provides the programmer with a mechanism for creating attributes specific to a program being written.

The attribute acts as storage for multiple argument values. Each attribute argument can be of any size or data type. The number of arguments, the argument size, and the data type are all specified here.

The attributes can be either created on the main object (object-level) or created on any of the object's data segments (segment-level). Whether the attribute is created at the object-level or at the segment-level is determined by the segment argument. If the segment argument is NULL, then the attribute will be instantiated at the object-level. Otherwise, the attribute will be instantiated on the specified segment, if it exists. If an attribute with the same name and scope already exists, then this function will fail.

The attribute can be either permanent or transient. Permanent implies that when the object is closed, a representation of the attribute will be written out to the disk.

In addition to being created either at the object or segment level, the attribute can also be created at the physical or presentation level.

Changes to physical level attributes are visible to all references of the data object and are termed as "shared" attributes. Changes to presentation level attributes are visible only to the reference object on which the change was made, and thus the attribute is termed as "unshared". Since all references may have potentially different values, unshared attributes may not be permanent.

An initial value must be provided. This initial value must consist of the proper number of arguments where each argument is of the same size and data type specified in this call.

The attribute can be destroyed with the `kdms_destroy_attribute` function.

Restrictions

This attribute will override any defined attributes with the same name and scope.

D.4.6. `kdms_destroy_attribute()` — *destroy an attribute*

Synopsis

```
int kdms_destroy_attribute(  
  
    kobject  object,  
    char     *segment,  
    char     *attribute)
```

Input Arguments

`object`
the object with the attribute to destroy

`segment`
the segment which contains the attribute. If this is NULL, then the attribute is assumed to exist at the object level.

`attribute`
string identifying the name of the attribute to destroy.

Returns

TRUE on success, FALSE otherwise

Description

This routine provides the programmer with a mechanism for destroying an instantiation of an attribute.

The segment argument is used to indicate which segment the attribute exists in. If the segment argument is NULL, then the attribute is assumed to exist at the object level.

If the attribute is not physically instantiated on the object, then it can not be destroyed. In general, this routine should only be used to destroy attributes that have been explicitly created with a `kdms_create_attribute` call.

D.4.7. `kdms_vset_attribute()` — *open varargs set attribute*

Synopsis

```
int  
kdms_vset_attribute(  
    kobject  object,
```

```
char    *segment,  
char    *attribute,  
kva_list *list)
```

Input Arguments

`object`
object to set the attribute on

`segment`
name of segment to set the attribute on.

`attribute`
name of the attribute to set `va_list` - a valist containing the values of the attribute.

Input:

Returns

TRUE on success, FALSE otherwise

Description

This routine allows the programmer to set the value of an attribute associated with a data object.

Data services manages two sets of attributes with each object. Internally, they are referred to as the physical attributes and the presentation attributes. The presentation version of an attribute is the value that is desired by the programmer. The corresponding physical attribute is what is physically stored on the file or transport associated with the object. If the two versions of an attribute result in different presentations of data, then data services automatically translates the data from the physical interpretation to the presentation interpretation, or visa-versa, as appropriate. For example, if the data type attribute (KDMS_DATA_TYPE) is physically unsigned int (KULONG), but the presentation value of that attribute is float (KFLOAT), then data services will retrieve the data from the object and cast it to float before returning it to the user. Similar behavior will occur for attributes such as index order (KDMS_INDEX_ORDER) and size (KDMS_SIZE).

The presentation attributes are settable at any time throughout a object's lifetime. The physical attributes, however, at some point are locked and thereafter cannot change. An object that was opened as an input object (via `kdms_input` or `kdms_open`) has its physical attributes locked immediately.

The effect of setting the attribute is immediate. For example, if this routine is used to set the KDMS_DATATYPE attribute, then any future access of the data in this image will involve data of the specified data type.

If the segment argument is NULL, then this instructs `kdms_set_attribute` to set a global attribute, if available. In either case, if the attribute does not exist, then this routine returns FALSE, indicating a failure.

D.4.8. `kdms_vset_attributes()` — *set attributes on a kvalist*

Synopsis

```
int
kdms_vset_attributes(
    kobject  object,
    char     *segment,
    kva_list *list)
```

Input Arguments

`object`
object to set the attribute on

`segment`
name of segment to set the attribute on. `va_list` - a valist containing attributes and their values.

Returns

TRUE on success, FALSE otherwise

Description

This routine allows the programmer to set the values of multiple attributes associated with a data object.

Data services manages two sets of attributes with each object. Internally, they are referred to as the physical attributes and the presentation attributes. The presentation version of an attribute is the value that is desired by the programmer. The corresponding physical attribute is what is physically stored on the file or transport associated with the object. If the two versions of an attribute result in different presentations of data, then data services automatically translates the data from the physical interpretation to the presentation interpretation, or visa-versa, as appropriate. For example, if the data type attribute (`KDMS_DATA_TYPE`) is physically unsigned int (`KULONG`), but the presentation value of that attribute is float (`KFLOAT`), then data services will retrieve the data from the object and cast it to float before returning it to the user. Similar behavior will occur for attributes such as index order (`KDMS_INDEX_ORDER`) and size (`KDMS_SIZE`).

The presentation attributes are settable at any time throughout a object's lifetime. The physical attributes, however, at some point are locked and thereafter cannot change. An object that was opened as an input object (via `kdms_input` or `kdms_open`) has its physical attributes locked immediately.

The effect of setting the attribute is immediate. For example, if this routine is used to set the `KDMS_DATATYPE` attribute, then any future access of the data in this image will involve data of the specified data type.

If the segment argument is `NULL`, then this instructs `kdms_set_attribute` to set a global attribute, if available. In either case, if the attribute does not exist, then this routine returns `FALSE`, indicating a failure.

D.4.9. `kdms_set_attribute()` — *set the value of an attribute*

Synopsis

```
int
kdms_set_attribute(
    kobject object,
    char *segment,
    char *attribute,
    kvalist)
```

Input Arguments

`object`

the object that is involved in the set attribute operation.

`segment`

the data segment whose attribute is being set. `va_alist` - variable argument list, that contains an attribute followed by any values associated with that attribute. It takes the form:

```
ATTRIBUTE_NAME, value1 [, value2, ...]
```

The number of value arguments in the variable argument list depends on the specific attribute. For example, `KDMS_DATATYPE` takes only one value, but `KDMS_SIZE` takes five values.

Returns

TRUE on success, FALSE otherwise

Description

This routine allows the programmer to set the value of an attribute associated with a data object.

Data services manages two sets of attributes with each object. Internally, they are referred to as the physical attributes and the presentation attributes. The presentation version of an attribute is the value that is desired by the programmer. The corresponding physical attribute is what is physically stored on the file or transport associated with the object. If the two versions of an attribute result in different presentations of data, then data services automatically translates the data from the physical interpretation to the presentation interpretation, or visa-versa, as appropriate. For example, if the data type attribute (`KDMS_DATA_TYPE`) is physically unsigned int (`KULONG`), but the presentation value of that attribute is float (`KFLOAT`), then data services will retrieve the data from the object and cast it to float before returning it to the user. Similar behavior will occur for attributes such as index order (`KDMS_INDEX_ORDER`) and size (`KDMS_SIZE`).

The presentation attributes are settable at any time throughout a object's lifetime. The physical attributes, however, at some point are locked and thereafter cannot change. An object that was opened as an input object (via `kdms_input` or `kdms_open`) has its physical attributes locked immediately.

The effect of setting the attribute is immediate. For example, if this routine is used to set the `KDMS_DATATYPE` attribute, then any future access of the data in this image will involve data of the

specified data type.

If the segment argument is NULL, then this instructs kdms_set_attribute to set a global attribute, if available. In either case, if the attribute does not exist, then this routine returns FALSE, indicating a failure.

D.4.10. kdms_set_attributes() — *sets the values of multiple attributes*

Synopsis

```
int
kdms_set_attributes(
    kobject object,
    char *segment,
    kvalist)
```

Input Arguments

object

the object that is involved in the set attribute operation.

segment

the data segment whose attribute is being set. va_alist - variable argument list, that contains a set of attributes, each followed by any values associated with that attribute. It takes the form:

```
ATTRIBUTE_NAME1, value1 [, value2, ...], ATTRIBUTE_NAME2, value1[, value2, ...],
```

The number of value arguments in variable argument list for each attribute depends on the specific attribute. For example, KDMS_DATATYPE takes only one value, but KDMS_SIZE takes five values. The NULL at the end of the variable argument list serves as a flag indicating the end of the list to kdms_set_attributes.

Returns

TRUE on success, FALSE otherwise

Description

The purpose of this routine is to set the values of multiple attributes of a data object.

Data services manages two sets of attributes with each object. Internally, they are referred to as the physical attributes and the presentation attributes. The presentation version of an attribute is the value that is desired by the programmer. The corresponding physical attribute is what is physically stored on the file or transport associated with the object. If the two versions of an attribute result in different presentations of data, then data services automatically translates the data from the physical interpretation to the presentation interpretation, or visa-versa, as appropriate. For example, if the data type attribute (KDMS_DATA_TYPE) is physically unsigned int (KULONG), but the presentation value of that attribute is float (KFLOAT), then data services will retrieve the data from the object and and cast it to

float before returning it to the user. Similar behavior will occur for attributes such as index order (KDMS_INDEX_ORDER) and size (KDMS_SIZE).

The presentation attributes are settable at any time throughout a object's lifetime. The physical attributes, however, at some point are locked and thereafter cannot change. An object that was opened as an input object (via `kdms_input` or `kdms_open`) has its physical attributes locked immediately.

The effect of setting an attribute is immediate. For example, if this routine is used to set the `KDMS_DATA_TYPE` attribute, then any future access of the data in this image will involve data of the specified data type.

If the segment argument is `NULL`, then this instructs `kdms_set_attributes` to get a series of global attributes, if available. In either case, if an attribute does not exist, then this routine returns `FALSE`, indicating a failure.

D.4.11. `kdms_vget_attribute()` — *get a single attribute on a kvalist*

Synopsis

```
int
kdms_vget_attribute(
    kobject  object,
    char     *segment,
    char     *attribute,
    kva_list *list)
```

Returns

TRUE on success, FALSE otherwise

Description

The purpose of this routine is to allow the programmer to get the value of an attribute associated with a data object.

Data services manages two sets of attributes with each object. Internally, they are referred to as the physical attributes and the presentation attributes. The presentation version of an attribute is the value that is desired by the programmer. The corresponding physical attribute is what is physically stored on the file or transport associated with the object. If the two versions of an attribute result in different presentations of data, then data services automatically translates the data from the physical interpretation to the presentation interpretation, or visa-versa, as appropriate. For example, if the data type attribute (`KDMS_DATA_TYPE`) is physically unsigned int (`KULONG`), but the presentation value of that attribute is float (`KFLOAT`), then data services will retrieve the data from the object and and cast it to float before returning it to the user. Similar behavior will occur for attributes such as index order (`KDMS_INDEX_ORDER`) and size (`KDMS_SIZE`).

The presentation attributes are settable at any time throughout a object's lifetime. The physical

attributes, however, at some point are locked and thereafter cannot change. An object that was opened as an input object (via `kdms_input` or `kdms_open`) has its physical attributes locked immediately.

If the `segment` argument is `NULL`, then this instructs `kdms_get_attribute` to get a global attribute, if available. In either case, if the attribute does not exist, then this routine returns `FALSE`, indicating a failure.

D.4.12. `kdms_vget_attributes()` — *get attributes on a kvalist*

Synopsis

```
int
kdms_vget_attributes(
    kobject  object,
    char     *segment,
    kva_list *list)
```

Returns

`TRUE` on success, `FALSE` otherwise

Description

The purpose of this routine is to allow the programmer to get the values of multiple attributes associated with a data object

Data services manages two sets of attributes with each object. Internally, they are referred to as the physical attributes and the presentation attributes. The presentation version of an attribute is the value that is desired by the programmer. The corresponding physical attribute is what is physically stored on the file or transport associated with the object. If the two versions of an attribute result in different presentations of data, then data services automatically translates the data from the physical interpretation to the presentation interpretation, or visa-versa, as appropriate. For example, if the data type attribute (`KDMS_DATA_TYPE`) is physically unsigned int (`KULONG`), but the presentation value of that attribute is float (`KFLOAT`), then data services will retrieve the data from the object and cast it to float before returning it to the user. Similar behavior will occur for attributes such as index order (`KDMS_INDEX_ORDER`) and size (`KDMS_SIZE`).

The presentation attributes are settable at any time throughout a object's lifetime. The physical attributes, however, at some point are locked and thereafter cannot change. An object that was opened as an input object (via `kdms_input` or `kdms_open`) has its physical attributes locked immediately.

If the `segment` argument is `NULL`, then this instructs `kdms_set_attributes` to get a series of global attributes, if available. In either case, if an attribute does not exist, then this routine returns `FALSE`, indicating a failure.

D.4.13. kdms_get_attribute() — *get the value of an attribute within a segment of an abstract object.*

Synopsis

```
int
kdms_get_attribute(
    kobject object,
    char *segment,
    char *attribute,
    kva_list)
```

Input Arguments

`object`

the object that is involved in the get attribute operation.

`segment`

the data segment whose attribute is being retrieved. `va_alist` - variable argument list, that contains an attribute followed by any addresses that will be filled out with the values associated with that attribute. The variable argument list takes the form:

```
ATTRIBUTE_NAME, &value1 [, &value2, ...]
```

The number of value arguments in variable argument list depends on the specific attribute. For example, `KDMS_DATATYPE` requires one value, but `KDMS_SIZE` requires five values.

Returns

TRUE on success, FALSE otherwise

Description

The purpose of this routine is to allow the programmer to get the value of an attribute associated with a data object.

Data services manages two sets of attributes with each object. Internally, they are referred to as the physical attributes and the presentation attributes. The presentation version of an attribute is the value that is desired by the programmer. The corresponding physical attribute is what is physically stored on the file or transport associated with the object. If the two versions of an attribute result in different presentations of data, then data services automatically translates the data from the physical interpretation to the presentation interpretation, or visa-versa, as appropriate. For example, if the data type attribute (`KDMS_DATA_TYPE`) is physically unsigned int (`KULONG`), but the presentation value of that attribute is float (`KFLOAT`), then data services will retrieve the data from the object and cast it to float before returning it to the user. Similar behavior will occur for attributes such as index order (`KDMS_INDEX_ORDER`) and size (`KDMS_SIZE`).

The presentation attributes are settable at any time throughout a object's lifetime. The physical attributes, however, at some point are locked and thereafter cannot change. An object that was opened as an input object (via `kdms_input` or `kdms_open`) has its physical attributes locked immediately.

If the segment argument is NULL, then this instructs kdms_get_attribute to get a global attribute, if available. In either case, if the attribute does not exist, then this routine returns FALSE, indicating a failure.

D.4.14. kdms_get_attributes() — *gets the values of a variable number of attributes within a single segment of an object.*

Synopsis

```
int
kdms_get_attributes(
    kobject object,
    char *segment,
    kvalist)
```

Input Arguments

object

the object that is involved in the set attribute operation.

segment

the data segment whose attribute is being retrieved. va_alist - a variable argument list that contains a set attributes, each followed by addresses to variables that will be assigned with values associated with that attribute. The variable argument list takes the form:

```
ATTRIBUTE_NAME1, &value1 [, &value2, ...], ATTRIBUTE_NAME2, &value1[, &value2, ...],
```

The number of value arguments in the variable argument list for each attribute depends on the specific attribute. For example, KDMS_DATATYPE takes only one value, but KDMS_SIZE takes multiple values. The NULL at the end of the variable argument list serves as a flag indicating the end of the list to kdms_get_attributes.

Returns

TRUE on success, FALSE otherwise

Description

The purpose of this routine is to allow the programmer to get the values of multiple attributes associated with a data object

Data services manages two sets of attributes with each object. Internally, they are referred to as the physical attributes and the presentation attributes. The presentation version of an attribute is the value that is desired by the programmer. The corresponding physical attribute is what is physically stored on the file or transport associated with the object. If the two versions of an attribute result in different presentations of data, then data services automatically translates the data from the physical interpretation to the presentation interpretation, or visa-versa, as appropriate. For example, if the data type attribute (KDMS_DATA_TYPE) is physically unsigned int (KULONG), but the presentation value of that

attribute is float (KFLOAT), then data services will retrieve the data from the object and cast it to float before returning it to the user. Similar behavior will occur for attributes such as index order (KDMS_INDEX_ORDER) and size (KDMS_SIZE).

The presentation attributes are settable at any time throughout a object's lifetime. The physical attributes, however, at some point are locked and thereafter cannot change. An object that was opened as an input object (via kdms_input or kdms_open) has its physical attributes locked immediately.

If the segment argument is NULL, then this instructs kdms_set_attributes to get a series of global attributes, if available. In either case, if an attribute does not exist, then this routine returns FALSE, indicating a failure.

Restrictions

Use only as directed...

D.4.15. kdms_match_attribute() — *returns TRUE if the same segment attribute in two abstract data objects match.*

Synopsis

```
int
kdms_match_attribute(
    kobject object1,
    kobject object2,
    char *segment,
    char *attribute)
```

Input Arguments

`object1`
the first abstract data object on which to match the specified attribute

`object2`
the second abstract data object on which to match the specified attribute

`segment`
the data segment in each abstract data object in which we will be matching the attribute.

`attribute`
the attribute that will be compared in the two objects.

Returns

There are three ways for this routine to return a FALSE: (1) if the attribute in the two objects does not match; (2) if either object does not contain the specified attribute; (3) an error condition resulting from an invalid object or segment. If none of these three conditions exist, then this function will return TRUE.

Description

The purpose of this routine is to allow the programmer to compare a single attribute between two data objects.

Data services manages two sets of attributes with each object. Internally, they are referred to as the physical attributes and the presentation attributes. The presentation version of an attribute is the value that is desired by the programmer. The corresponding physical attribute is what is physically stored on the file or transport associated with the object. If the two versions of an attribute result in different presentations of data, then data services automatically translates the data from the physical interpretation to the presentation interpretation, or visa-versa, as appropriate. For example, if the data type attribute (KDMS_DATA_TYPE) is physically unsigned int (KULONG), but the presentation value of that attribute is float (KFLOAT), then data services will retrieve the data from the object and cast it to float before returning it to the user. Similar behavior will occur for attributes such as index order (KDMS_INDEX_ORDER) and size (KDMS_SIZE).

The presentation attributes are settable at any time throughout a object's lifetime. The physical attributes, however, at some point are locked and thereafter cannot change. An object that was opened as an input object (via kdms_input or kdms_open) has its physical attributes locked immediately.

This routine will return TRUE if the specified attribute has the same value in both of the abstract data objects. This routine will return FALSE if the attribute does not have the same value in both of the objects kdms_match_attribute will also return FALSE if the attribute does not exist in either or both of the objects.

If the segment argument is NULL, then that implies that the attribute is a global attribute in each of the abstract data objects.

D.4.16. kdms_vmatch_attributes() — *returns true if the vararg list of segment attributes in two abstract data objects match.*

Synopsis

```
int
kdms_vmatch_attributes(
    kobject  object1,
    kobject  object2,
    char     *segment,
    kva_list *list)
```

Input Arguments

object1
the first abstract data object on which to match the specified attributes

object2
the second abstract data object on which to match the specified attributes

segment
the data segment in each abstract data object in which we will be matching the attributes.

list
variable argument list, that contains an arbitrarily long list of attributes followed a NULL. It takes the form:

ATTRIBUTE_NAME1, ATTRIBUTE_NAME2, ..., NULL

Returns

There are three ways for this routine to return a FALSE: (1) if any of the attributes between the two objects do not match; (2) if either object does not contain one or more of the specified attributes; (3) an error condition resulting from an invalid object or segment. If none of these three conditions exist, then this function will return TRUE.

Description

The purpose of this routine is to allow the programmer to compare multiple attributes in two object.

This routine will return TRUE if all of the specified attributes have the same value in the objects. This routine will return FALSE if any of the attributes do not match kdms_match_attributes will also return FALSE if any of the attributes do not exist in either or both of the two objects.

Data services manages two sets of attributes with each object. Internally, they are referred to as the physical attributes and the presentation attributes. The presentation version of an attribute is the value that is desired by the programmer. The corresponding physical attribute is what is physically stored on the file or transport associated with the object. If the two versions of an attribute result in different presentations of data, then data services automatically translates the data from the physical interpretation to the presentation interpretation, or visa-versa, as appropriate. For example, if the data type attribute (KDMS_DATA_TYPE) is physically unsigned int (KULONG), but the presentation value of that attribute is float (KFLOAT), then data services will retrieve the data from the object and and cast it to

float before returning it to the user. Similar behavior will occur for attributes such as index order (KDMS_INDEX_ORDER) and size (KDMS_SIZE).

The presentation attributes are settable at any time throughout a object's lifetime. The physical attributes, however, at some point are locked and thereafter cannot change. An object that was opened as an input object (via `kdms_input` or `kdms_open`) has its physical attributes locked immediately.

If the segment argument is NULL, then that implies that the attributes are global attributes in each of the abstract data objects.

D.4.17. `kdms_match_attributes()` — *returns true if the list of segment attributes in two abstract data objects match.*

Synopsis

```
int
kdms_match_attributes(
    kobject object1,
    kobject object2,
    char *segment,
    kvalist)
```

Input Arguments

`object1`

the first abstract data object on which to match the specified attributes

`object2`

the second abstract data object on which to match the specified attributes

`segment`

the data segment in each abstract data object in which we will be matching the attributes. `va_alist` - variable argument list, that contains an arbitrarily long list of attributes followed a NULL. It takes the form:

```
ATTRIBUTE_NAME1, ATTRIBUTE_NAME2, ..., NULL
```

Returns

There are three ways for this routine to return a FALSE: (1) if any of the attributes between the two objects do not match; (2) if either object does not contain one or more of the specified attributes; (3) an error condition resulting from an invalid object or segment. If none of these three conditions exist, then this function will return TRUE.

Description

The purpose of this routine is to allow the programmer to compare multiple attributes in two object.

This routine will return TRUE if all of the specified attributes have the same value in the objects. This

routine will return FALSE if any of the attributes do not match `kdms_match_attributes` will also return FALSE if any of the attributes do not exist in either or both of the two objects.

Data services manages two sets of attributes with each object. Internally, they are referred to as the physical attributes and the presentation attributes. The presentation version of an attribute is the value that is desired by the programmer. The corresponding physical attribute is what is physically stored on the file or transport associated with the object. If the two versions of an attribute result in different presentations of data, then data services automatically translates the data from the physical interpretation to the presentation interpretation, or visa-versa, as appropriate. For example, if the data type attribute (`KDMS_DATA_TYPE`) is physically unsigned int (`KULONG`), but the presentation value of that attribute is float (`KFLOAT`), then data services will retrieve the data from the object and cast it to float before returning it to the user. Similar behavior will occur for attributes such as index order (`KDMS_INDEX_ORDER`) and size (`KDMS_SIZE`).

The presentation attributes are settable at any time throughout a object's lifetime. The physical attributes, however, at some point are locked and thereafter cannot change. An object that was opened as an input object (via `kdms_input` or `kdms_open`) has its physical attributes locked immediately.

If the segment argument is NULL, then that implies that the attributes are global attributes in each of the abstract data objects.

D.4.18. `kdms_copy_attribute()` — *copy an attribute from a source object to a destination object.*

Synopsis

```
int
kdms_copy_attribute(
    kobject object1,
    kobject object2,
    char *segment,
    char *attribute)
```

Input Arguments

`object1`
the source for the copy operation

`object2`
the destination for the copy operation

`segment`
the segment which contains the object to be copied.

`attribute`
the attribute to be copied.

Returns

TRUE on success, FALSE otherwise.

Description

This function is used to copy a single attribute from one object to another object.

Data Services manages two versions of some of the attributes associated with each object. These attributes are the size and data type. Internally, the two versions of these attributes are referred to as the physical attribute and the presentation attribute. Typically, the programmer has access to only the presentation versions of these attributes. The physical attributes are set indirectly depending on the setting of KPDS_VALUE_COUPLING. See `kpds_get_data` for a description of how the presentation and physical attributes affect interaction with the data object.

Other attributes are classified as either shared or unshared. Shared attributes are stored at the physical layer of the attribute, and thus can be shared by multiple references of the data object (see `kpds_reference_object` for more information about references). Unshared attributes, on the other hand, can only be used by the local object (see `kpds_query_attribute` for information on how to determine whether an attribute is shared or unshared).

This function accepts a source object, a destination object, and an attribute name. If the attribute exists in the source object, then it will be copied to the destination object. If the attribute does not exist in the source object, then an error condition is returned.

D.4.19. `kdms_vcopy_attributes()` — *copy attributes given in a kvalist*

Synopsis

```
int
kdms_vcopy_attributes(
    kobject  object1,
    kobject  object2,
    char     *segment,
    kva_list *list)
```

Input Arguments

`object1`
the source for the copy operation

`object2`
the destination for the copy operation

`segment`
the segment which contains the object to be copied. `kvalist` - A NULL terminated variable argument list of attributes to be copied.

Returns

TRUE on success, FALSE otherwise.

Description

This function is used to copy multiple attributes given a variable argument list containing a list attributes to copy.

Data Services manages two versions of some of the attributes associated with each object. These attributes are the size and data type. Internally, the two versions of these attributes are referred to as the physical attribute and the presentation attribute. Typically, the programmer has access to only the presentation versions of these attributes. The physical attributes are set indirectly depending on the setting of KPDS_VALUE_COUPLING. See `kpds_get_data` for a description of how the presentation and physical attributes affect interaction with the data object.

Other attributes are classified as either shared or unshared. Shared attributes are stored at the physical layer of the attribute, and thus can be shared by multiple references of the data object (see `kpds_reference_object` for more information about references). Unshared attributes, on the other hand, can only be used by the local object (see `kpds_query_attribute` for information on how to determine whether an attribute is shared or unshared).

This function accepts a source object, a destination object, and an attribute name. If the attribute exists in the source object, then it will be copied to the destination object. If the attribute does not exist in the source object, then an error condition is returned.

D.4.20. `kdms_copy_attributes()` — *copy attributes from a source object to a destination object.*

Synopsis

```
int
kdms_copy_attributes(
    kobject object1,
    kobject object2,
    char *segment,
    kvalist)
```

Input Arguments

`object1`
the source for the copy operation

`object2`
the destination for the copy operation

`segment`
the segment which contains the object to be copied.

`kvalist`
A NULL terminated variable argument list of attributes to be copied.

Returns

TRUE on success, FALSE otherwise.

Description

This function is used to copy multiple attributes given a variable argument list containing a list attributes to copy.

Data Services manages two versions of some of the attributes associated with each object. These attributes are the size and data type. Internally, the two versions of these attributes are referred to as the physical attribute and the presentation attribute. Typically, the programmer has access to only the presentation versions of these attributes. The physical attributes are set indirectly depending on the setting of KPDS_VALUE_COUPLING. See `kpds_get_data` for a description of how the presentation and physical attributes affect interaction with the data object.

Other attributes are classified as either shared or unshared. Shared attributes are stored at the physical layer of the attribute, and thus can be shared by multiple references of the data object (see `kpds_reference_object` for more information about references). Unshared attributes, on the other hand, can only be used by the local object (see `kpds_query_attribute` for information on how to determine whether an attribute is shared or unshared).

This function accepts a source object, a destination object, and an attribute name. If the attribute exists in the source object, then it will be copied to the destination object. If the attribute does not exist in the source object, then an error condition is returned.

D.4.21. `kdms_query_attribute()` — *get information about an attribute*

Synopsis

```
int
kdms_query_attribute(
    kobject object,
    char *segment,
    char *attribute,
    int *numargs,
    int *argsize,
    int *datatype,
    int *permanent)
```

Input Arguments

`object`
the object with the attribute being queried

`segment`
the segment that the attribute is stored in. If this argument is NULL, then the attribute is global to the object. `name` - name of the attribute to be queried.

`numargs`
number of arguments in this attribute

`argsize`
size of each argument in this attribute.

`datatype`

datatype of the attribute
permanent
is the attribute stored or transient? The return value will be either TRUE or FALSE

Returns

TRUE if attribute exists, FALSE otherwise

Description

This function is used for two purposes: (1) to determine the existence of an attribute; and (2) to obtain the characteristics of the attribute.

Data Services manages two versions of some of the attributes associated with each object. These attributes are the size and data type. Internally, the two versions of these attributes are referred to as the physical attribute and the presentation attribute. Typically, the programmer has access to only the presentation versions of these attributes. The physical attributes are set indirectly depending on the setting of KDMS_COUPLING. See `kpds_get_data` for a description of how the presentation and physical attributes affect interaction with the data object.

Other attributes are classified as either shared or unshared. Shared attributes are stored at the physical layer of the attribute, and thus can be shared by multiple references of the data object (see `kpds_reference_object` for more information about references). Unshared attributes, on the other hand, can only be used by the local object.

The difference between shared and unshared attributes is abstracted from the user at the PDS level. The permanent attributes are generally shared, and the non-permanent attributes are generally non-shared. Permanent attributes are attributes that will be stored as part of an output object when it is written. Any attributes that are retrieved when an object is opened are also permanent attributes. Non-permanent attributes exist only while the program that is operating on the object is executing.

The datatype argument indicates what kind of information is stored in the attribute. Attributes can be one of the following data types: KBYTE, KUBYTE, KSHORT, KUSHORT, KINT, KUINT, KLONG, KULONG, KFLOAT, KDOUBLE, KCOMPLEX, or KDCOMPLEX.

The numargs argument indicates how many arguments must be passed in an argument list to one of the attribute functions.

The argsize arguments indicates the number of units of the data type there are in each argument. This argument allows arrays of information to be stored as attributes.

D.4.22. `kdms_print_attribute()` — *print the value of an attribute*

Synopsis

int

```
kdms_print_attribute(  
    kobject object,  
    char *segment,  
    char *attribute,  
    kfile *printfile)
```

Input Arguments

`object`
the object containing the attribute

`segment`
the segment in the object which contains the attribute.

`attribute`
the attribute to print

`printfile`
the open kfile to print to

Returns

TRUE on success, FALSE otherwise

Description

This function is used to print a single attribute to an open kfile.

This function is typically used by such programs as `kprdata` to print out the values of attributes in an object.

D.4.23. <code>kdms_get_attribute_names()</code> — <i>get a list of attributes from an object.</i>
--

Synopsis

```
char **  
kdms_get_attribute_names(  
    kobject object,  
    char *segment,  
    char *filter,  
    int permanent,  
    int *number)
```

Input Arguments

`object`
the object to get the attribute names from.

`segment`
the segment to get the attribute names from. If NULL, then get the object-level names.

`filter`

a regular expression filter. Only attribute names passing the filter will be returned in the list.

`permanent`

TRUE if only permanent attributes should be included in the list

Output Arguments

`number`

the number of attribute names that are being returned

Returns

an array of attribute names

Description

This function returns a list of attributes associated with the specified segment. If the segment is NULL, then it returns a list of the attributes for the object.

The allowable regular expression syntax is :

`.` Match any single character except newline

`*` Match the preceding character or range of characters 0 or more times. The matching includes items within a [...].

`[...]` or `[^..]` Matches any one character contained within the brackets. If the first character after the '[' is the ']', then it is included in the characters to match. If the first character after the '[' is a '^', then it will match all characters NOT included in the []. The '-' will indicate a range of characters. For example, `[a-z]` specifies all characters between and including the ascii values 'a' and 'z'. If the '-' follows the '[' or is right before the ']' then it is interpreted literally.

`^` If this is the first character of the regular expression, it matches the beginning of the line.

`$` If this is the last character of the regular expression, it matches the end of the line.

\ This escapes the meaning of a special character.

The array that is returned must be freed by the user using the call `karray_free`.

D.5. Interactivity Management

- `kdms_add_callback()` - add a callback associated with an object's data or attribute.
- `kdms_remove_callback()` - remove a callback associated with an object's data or attribute.

D.5.1. `kdms_add_callback()` — *add a callback associated with an object's data or attribute.*

Synopsis

```
int
kdms_add_callback(
    kobject    object,
    char       *segment,
    char       *type,
    kfunc_void callfunc,
    kaddr      clientData)
```

Input Arguments

`object`
The object to add callback to.

`segment`
The segment in object to add callback to. If NULL, then it is considered an object callback.

`type`
The type of callback to add.

`callfunc`
The function to call when the callback occurs.

`clientData`
Any clientdata to be passed to the callfunc.

Returns

TRUE on success, FALSE otherwise

Description

This function is used to add a callback on a segment in a data object so that operations can be performed and monitored in an event driven environment.

A callback is a mechanism for operating on a data object whenever an asynchronous event occurs rather than in a strictly sequential manner. This function is particularly useful in conjunction with `kdms_reference`. The callbacks operate on the data, which is shared between referenced objects. Thus object A can set a callback such that when data get changed via object B, operations can be performed on object A.

The object and segment arguments are used to specify which segment and object will have the callback added to.

The type argument specifies what kind of callback should be added. This argument may take on the

following values:

- KDMS_CALLBACK_CHANGE - generate a callback whenever data on the specified segment is changed (via `kdms_put_data` or `kdms_copy_data`).
- KDMS_CALLBACK_ACCESS - generate a callback whenever data on the specified segment is accessed (via `kdms_get_data` or `kdms_copy_data`).
- KDMS_CALLBACK_DELETE - generate a callback whenever the segment specified is about to be deleted, either through a `kdms_destroy_segment()` call, or via a `kdms_close()`.
- KDMS_CALLBACK_DESTROY - generate a callback whenever the last instance of an object or segment is about to be closed.
- KDMS_CALLBACK_SAVE - generate a callback whenever the specified segment is about to be changed. This is similar to the `KDMS_CALLBACK_CHANGE`, except that the callback is generated before the data is changed rather than afterward.

If the segment specified is `NULL`, then the callback is placed on the entire object. Currently, only `DELETE` and `DESTROY` callbacks can be placed on an object.

The callback mechanisms in Data Services are analogous to the callback mechanisms that are available in the `Xvwidget` library or in `Xt`. The motivation for callbacks in Data Services is that they facilitate functionality in the `Xvisual` library, where they allow different visual objects tied to a single data object to communicate with each other.

Restrictions

Only `KDMS_CALLBACK_DELETE` and `KDMS_CALLBACK_DESTROY` callbacks work for objects.

D.5.2. `kdms_remove_callback()` — *remove a callback associated with an object's data or attribute.*

Synopsis

```
int
kdms_remove_callback(
    kobject    object,
```

```
char      *segment,  
char      *type,  
kfunc_void callfunc,  
kaddr     clientData)
```

Input Arguments

`object`
object to add callback to.

`segment`
segment in object to add callback to.

`type`
type of callback to add.

`callfunc`
function to call when callback occurs.

`clientData`
data to be passed to callfunc.

Returns

TRUE on success, FALSE otherwise

Description

This function is used to remove a callback that was previously added on a segment in a data object so that operations can be performed and monitored in an event driven environment.

This function will not remove a callback unless all of the arguments that are passed to it are the same as those passed to the `kdms_create_callback` previously. This allows for multiple callbacks that are similar in nature to be removed without confusion.

A callback is a mechanism for operating on a data object whenever an asynchronous event occurs rather than in a strictly sequential manner. This function is particularly useful in conjunction with `kdms_reference`. The callbacks operate on the data, which is shared between referenced objects. Thus object A can set a callback such that when data get changed via object B, operations can be performed on object A.

The `object` and `segment` arguments are used to specify which segment and object will have the callback added to.

The `type` argument specifies what kind of callback should be added. This argument may take on the following values:

`KDMS_CALLBACK_CHANGE` - generate a callback whenever data on the specified segment is changed (via `kdms_put_data` or `kdms_copy_data`).

`KDMS_CALLBACK_ACCESS` - generate a callback whenever data on the specified segment is accessed (via `kdms_get_data` or `kdms_copy_data`).

`KDMS_CALLBACK_DELETE` - generate a callback whenever the

segment specified is about to be deleted, either through a `kdms_destroy_segment()` call, or via a `kdms_close()`.

`KDMS_CALLBACK_DESTROY` - generate a callback whenever the last instance of an object or segment is about to be closed.

`KDMS_CALLBACK_SAVE` - generate a callback whenever the specified segment is about to be changed. This is similar to the `KDMS_CALLBACK_CHANGE`, except that the callback is generated before the data is changed rather than afterward.

If the segment specified is `NULL`, then the callback is placed on the entire object. Currently, only `DELETE` and `DESTROY` callbacks can be placed on an object.

The callback mechanisms in Data Services are analogous to the callback mechanisms that are available in the `Xvwidget` library or in `Xt`. The motivation for callbacks in Data Services is that they facilitate functionality in the `Xvisual` library, where they allow different visual objects tied to a single data object to communicate with each other.

Restrictions

Only `KDMS_CALLBACK_DELETE` and `KDMS_CALLBACK_DESTROY` callbacks work for objects.

D.6. Data Manipulation

- `kdms_get_data()` - get data from data object
- `kdms_put_data()` - put data into object
- `kdms_copy_remaining_data()` - copy remaining data

D.6.1. `kdms_get_data()` — *get data from data object*

Synopsis

```
kaddr
kdms_get_data(
    kobject object,
    char *segment,
    int *begin,
    int *end,
    kaddr data)
```

Input Arguments

`object`
the object from which the data will be obtained.

`segment`
the segment from which the data will be obtained.

`begin`
the begin marker of the region of data to be retrieved.

`end`
the end marker of the region of data to be retrieved.

`data`
a pointer to the region of memory that will serve as a destination for the data. If this value is NULL, then sufficient space for this operation will be allocated for this operation. The data type `kaddr` is used because it indicates a generic data pointer.

Returns

If "data" is not initially NULL, then the data space pointed to by "data" will be returned on success. If the "data" argument is NULL, then a new pointer to the requested data will be returned. Unsuccessful calls to this routine are indicated by a return value of NULL.

Description

`kdms_get_data` is used to obtain data that is stored in a data object. The data that is retrieved is designated by two "corner-markers". These are arrays which contain N integer values, where N is the dimensionality of the segment (the dimensionality of a segment can be determined with the `KDMS_DIMENSION` attribute). All values in begin argument must be less than or equal to their corresponding value in the end argument. In a two dimensional case, the begin marker is the upper left corner and the end marker is the lower right corner of a rectangle that is obtained with this function call. The coordinate origin is the upper, left, front corner of the data set extended to N-space. The corner markers are specified in the index order that the data set is presented in.

Restrictions

This routine assumes that if the argument "data" is not NULL, then it contains the appropriate amount of memory with the appropriate dimensionality for the requested primitive.

D.6.2. `kdms_put_data()` — *put data into object*

Synopsis

```
int
kdms_put_data(
    kobject object,
    char    *segment,
    int     *begin,
    int     *end,
```

```
kaddr data)
```

Input Arguments

`object`
the object from which the data will be stored.

`segment`
the segment from which the data will be stored.

`begin`
the begin marker of the region of data to be retrieved.

`end`
the end marker of the region of data to be retrieved.

`data`
a pointer to the region of memory that will serve as a

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

`kdms_put_data` is used to store data that into a data object. The data that is stored is designated by two "corner-markers". These are arrays which contain N integer values, where N is the dimensionality of the segment (the dimensionality of a segment can be determined with the `KDMS_DIMENSION` attribute). All values in `begin` argument must be less than or equal to their corresponding value in the `end` argument. In a two dimensional case, the `begin` marker is the upper left corner and the `end` marker is the lower right corner of a rectangle that is obtained with this function call. The coordinate origin is the upper, left, front corner of the data set extended to N-space. The corner markers are specified in the index order that the data set is presented in.

D.6.3. `kdms_copy_remaining_data()` — *copy remaining data*

Synopsis

```
int  
kdms_copy_remaining_data(  
    kobject subject,  
    kobject dobject)
```

Input Arguments

`subject`
object to get data from

`dobject`
object to copy data to

Description

kdms_copy_remaining_data copies any data from segments which are instantiated in both the source and destination objects and have not yet had their data modified in the destination.

This page left intentionally blank

Table of Contents

A. Introduction	5-1
B. Presentation of Data †	5-2
B.1. Casting	5-3
B.2. Scaling and Normalization	5-3
B.3. Padding and Interpolation	5-4
B.4. Conversion of Complex Data	5-5
B.5. Index Order Manipulation	5-5
C. Attributes	5-7
C.1. Global Attributes	5-7
C.2. Segment Attributes	5-9
D. Functions Provided By Data Management Services	5-15
D.1. Object Management	5-15
D.1.1. kdms_create() — <i>create a temporary data object.</i>	5-15
D.1.2. kdms_open() — <i>create an object associated with an input or output transport.</i>	5-16
D.1.3. kdms_close() — <i>close an open data object.</i>	5-18
D.1.4. kdms_reopen() — <i>associate new data with an existing object</i>	5-18
D.1.5. kdms_reference() — <i>create a reference of a data object.</i>	5-19
D.1.6. kdms_sync() — <i>synchronize physical and presentation layers of a data object.</i>	5-20
D.1.7. kdms_update_references() — <i>update segment presentation of all reference objects.</i>	5-20
D.1.8. kdms_close_hook() — <i>insert a service to be called when an object is closed.</i>	5-21
D.1.9. kdms_reference_list() — <i>return a klist of references.</i>	5-22
D.1.10. kdms_get_segment_names() — <i>get an array of segment names for the object specified.</i>	5-22
D.2. Information	5-23
D.2.1. kdms_support() — <i>obtain a list of file formats supported by data services.</i>	5-23
D.3. Segment Management	5-23
D.3.1. kdms_query_segment() — <i>determine if a data segment is available.</i>	5-24
D.3.2. kdms_create_segment() — <i>create a segment on a data object.</i>	5-24
D.3.3. kdms_destroy_segment() — <i>destroy a segment from a data object.</i>	5-25
D.3.4. kdms_rename_segment() — <i>rename a segment</i>	5-25
D.4. Attribute Management	5-26
D.4.1. kdms_define_quasi_attribute() — <i>define a quasi attribute</i>	5-26
D.4.2. kdms_define_attribute() — <i>define an attribute for for a session</i>	5-31
D.4.3. kdms_undefine_attribute() — <i>undefine a defined attribute</i>	5-33
D.4.4. kdms_query_attribute_definition() — <i>determines if an attribute is defined.</i>	5-33
D.4.5. kdms_create_attribute() — <i>instantiate an attribute</i>	5-34
D.4.6. kdms_destroy_attribute() — <i>destroy an attribute</i>	5-36
D.4.7. kdms_vset_attribute() — <i>open varargs set attribute</i>	5-36
D.4.8. kdms_vset_attributes() — <i>set attributes on a kvalist</i>	5-38
D.4.9. kdms_set_attribute() — <i>set the value of an attribute</i>	5-39
D.4.10. kdms_set_attributes() — <i>sets the values of multiple attributes</i>	5-40
D.4.11. kdms_vget_attribute() — <i>get a single attribute on a kvalist</i>	5-41
D.4.12. kdms_vget_attributes() — <i>get attributes on a kvalist</i>	5-42
D.4.13. kdms_get_attribute() — <i>get the value of an attribute within a segment of an abstract object.</i>	5-43
D.4.14. kdms_get_attributes() — <i>gets the values of a variable number of attributes within a single segment of an object.</i>	5-44

D.4.15. kdms_match_attribute() — returns TRUE if the same segment attribute in two abstract data objects match.	5-45
D.4.16. kdms_vmatch_attributes() — returns true if the vararg list of segment attributes in two abstract data objects match.	5-47
D.4.17. kdms_match_attributes() — returns true if the list of segment attributes in two abstract data objects match.	5-48
D.4.18. kdms_copy_attribute() — copy an attribute from a source object to a destination object.	5-49
D.4.19. kdms_vcopy_attributes() — copy attributes given in a kvalist	5-50
D.4.20. kdms_copy_attributes() — copy attributes from a source object to a destination object.	5-51
D.4.21. kdms_query_attribute() — get information about an attribute	5-52
D.4.22. kdms_print_attribute() — print the value of an attribute	5-53
D.4.23. kdms_get_attribute_names() — get a list of attributes from an object.	5-54
D.5. Interactivity Management	5-57
D.5.1. kdms_add_callback() — add a callback associated with an object's data or attribute.	5-57
D.5.2. kdms_remove_callback() — remove a callback associated with an object's data or attribute.	5-58
D.6. Data Manipulation	5-60
D.6.1. kdms_get_data() — get data from data object	5-60
D.6.2. kdms_put_data() — put data into object	5-61
D.6.3. kdms_copy_remaining_data() — copy remaining data	5-62

Chapter 6

Structure Support

Chapter 6 - Structure Support

A. Overview

This document details the support provided in VisiQuest for the storage and retrieval of complex data structures. Data structures are defined within VisiQuest to be new "data types." This is similar to the notion of data types in C. Just as a variable could be declared as an *int* or a *float*, it could also be declared to be of some structure type. This notion, applied to Data Services, introduces the idea of *structure attributes*. One of the characteristics of an attribute is its data type. By creating a Data Services attribute to be of a defined structure data type, variables of that structure can be stored and retrieved from that attribute. This allows structures to be passed from one program to another via Data Services objects.

Structures are defined by creating a structure specification file. This file, by convention, has an *.x* extension. Once a specification has been created, a program called *kgenstruct* parses the file and generates two files. The first file generated is a C header file which contains the actual data structures to be used within your own C code. The second file generated is a C code file containing writer and reader routines which can flatten an instance of the defined structure from memory to a file, and unflatten the structure from the file back into memory. Routines for comparing and freeing the defined structure are also generated, along with a definition routine which initializes the structure as a new type in VisiQuest.

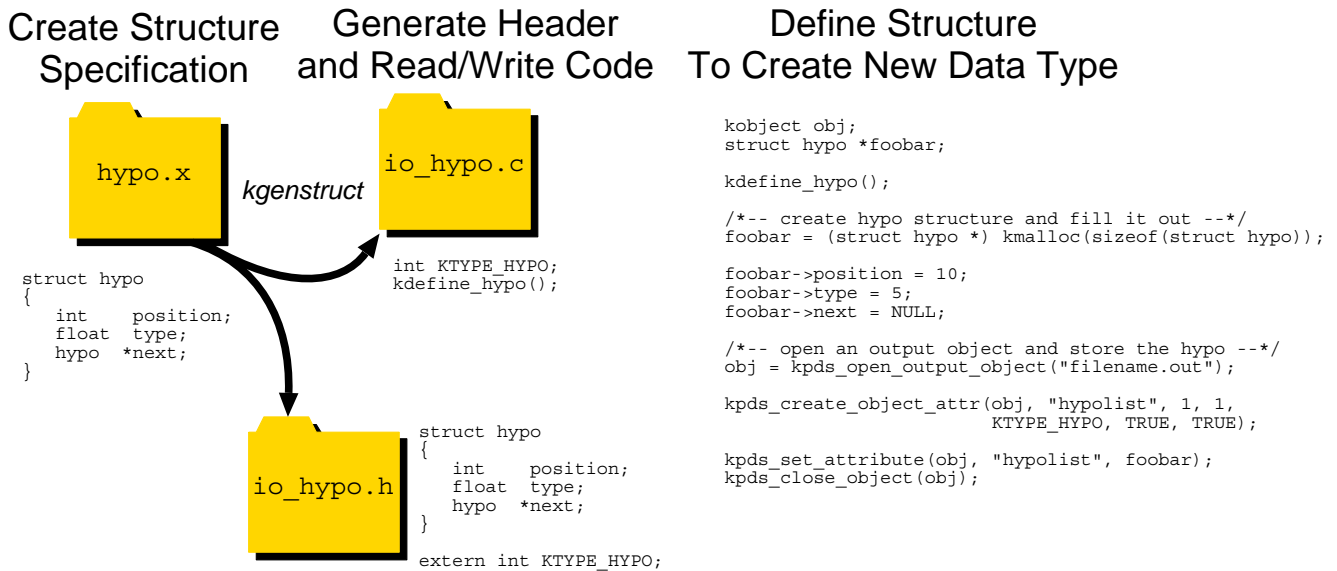


Figure 1: This picture overviews the generation of reader and writer code by *kgenstruct* from a structure spec file, and how the generated code is used to define a new data type in VisiQuest. The example illustrates an attribute of the defined structure type being created and set on a data object.

A structure is defined by internally associating the reader and writer routines with a type identifier. This type identifier is analogous to the VisiQuest type identifiers for standard C data types, such as *KINT* and *KFLOAT*. The type identifier and the routine which initializes it are generated for you by *kgenstruct*. By convention, the

type identifier has the same name as the structure, prepended by `KTYPE_`. Thus for a structure "matrix", the type identifier `KTYPE_MATRIX` will be created. This type identifier would be initialized by calling the definition routine `kdefine_matrix()`.

The generated routines read and write the structure in pieces, breaking the structure down by fields. Each field is written using the VisiQuest generic read and write calls. These calls provide the machine conversion capability necessary to write data in the native format for one architecture and convert the data while it is being read into the native format for a different architecture. Pointers are traversed on reading and writing using special pointer read and write calls. The write call manages the addresses of what is being pointed at, ensuring that items being pointed at are not written more than once. The read call then manages the addresses of what is being read, reconnecting pointers to restore the structure as it existed before being written. This capability allows for the storage and retrieval of very complicated structures with cyclic and redundant pointer dependencies.

B. Passing Structures Between Programs

Structures can be passed from one program to in several ways: either by writing the structure to and reading the structure from a VisiQuest transport, by storing the structures in a data services object, or by sending the structures down a data services stream.

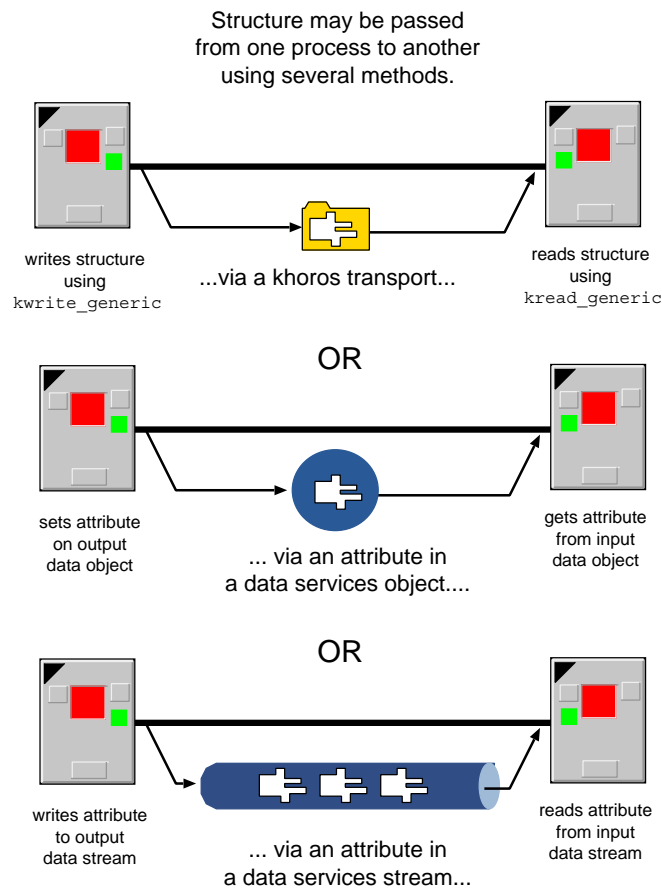


Figure 2: This figure illustrates the basic concept of how a structures may be passed from one routine to another in VisiQuest.

The transport mechanism is simple. The writing process simply opens a VisiQuest transport using the `kopen` function, and then writes the structure to the transport using the `kwrite_generic` function. The reading process then opens the transport and reads the structure using the `kread_generic` function.

The data object mechanism in Data Services allows the creation of attributes of any data type. For structures, the defined structure type identifier can be used to specify the data type of an attribute. To pass a structure from one program to another, a program simply open an output object, creates or defines an attribute of a structure type, and then stores the structure with a standard Data Services set attribute call. Other programs which also have the structure data type defined can then open the object and retrieve the attribute.

The data stream is similar to the data object paradigm, except that a Data Services stream is used instead of an object.

C. Structure Specification File

A structure is added as a new data type by specifying it within a *structure specification file*. A structure specification file will appear to be syntactically very much like a regular C header file. In fact, you can likely transfer any structures you may already be using directly from a header file into a specification file with little or no change. These files are different, however, from C header files in that they are never directly included by your C code. A code generator, `kgenstruct`, will parse the specification file and generate an actual C header which you should include. Additionally, `kgenstruct` will also generate a C source code file which you should either compile into your program, or compile into a library which other programs will link against.

C.1. Creating a Structure Specification File

A structure specification file is considered to be a source file in a Software Object. It can be created from the software object editor Composer by using the *Add File to Object* operation under the *File Operations* pulldown menu for SOURCE objects. The file you create should end with a `.x`. This extension indicates that the file is a structure specification file. Once created, the new specification file will be listed as one of the *Source Code* files in the file list, and can be edited using the *Edit* operation under the *File Operations* pulldown menu.

C.2. Kgenstruct

`kgenstruct` should be run once your specification file is completed. This program is best invoked as an operation in Composer. This operation is available as a "kmake struct" rule and can be accessed from the *Commands* menu, *on Console* option, *Make* button. Note that, if you are working outside of `composer`, it can also be run from the command line. When run from the command line, this program takes a *toolbox* and *object name* argument.

This program is a ported and heavily modified version of `rpcgen`. As such, it is capable of parsing nearly all of RPCL, a protocol description language which is an extension of the XDR definition language. It is likely that any structure specification files which you may have been using with `rpcgen` can be used directly with `kgenstruct` to migrate your code to VisiQuest.

C.3. What is Generated?

`kgenstruct` generates two files from the specification : a header file and a C code file. The naming convention for these files is to prepend `io_` to the name of the specification file. Thus, for a specification `matrix.x`, the files

io_matrix.c and io_matrix.h will be generated. The generated .h file should be included by any programs which need to use the structure. This is the actual C definition which you will write your source code to.

The generated .c file will contain several routines for handling each structure specified in the specification file. For each structure, a reader routine, a writer routine, a comparison routine and a free routine will be generated. A type handle to use to signify the use of the structure as a data type to other VisiQuest routines is also generated. This type handle will be named after the structure name in all capitals, prepended by `KTYPE_`. A definition routine which initializes the type identifier is also generated. This routine will be named after the structure name, prepended by `kdefine_` and will take no arguments. It should be called prior to any use of your structure with any VisiQuest routines.

The reader, writer, comparison, and free routines which are generated will be associated with the type identifier through the generated define routine. Once defined, the structure can be treated as a new data type through various foundation library routines. For example, the comparison and free routines can be accessed via the foundation routines `kstruct_compare()` and `kstruct_free()`. Additionally, the `kread_generic()` and `kwrite_generic()` calls will now understand the structure identifier and will be able to read and write structures, just as they now read and write other data types. It is not recommended that these routines be used directly for reading and writing structures. Data Services attributes are instead recommended for reading and writing structures.

One final note : always make changes to the specification file and then regenerate the header and code files. Never edit the generated files directly. Any changes you might make to the generated files will be overwritten and lost the next time you regenerate from your specification.

C.4. Software Object Types

If the specification file is part of a library object, the generated C code file will be generated in that library's source directory, and the generated header file will be added to the library's public include directory. This approach should be taken for structures which will be used by multiple programs. The structure specification and generated structure define routine will be in a common library. Each program needing to use the structures should include the library's include file and link against the library.

If the specification file is part of any other type of software object, the generated C code and header files will be generated in the software object's source directory. Note also that `kgenstruct` can be used on .x files outside of the context of VisiQuest, in which case the generated C code and header is generated in the local directory.

C.5. Specification Language

The *specification language* is, with few exceptions, very similar to C. For the most part, the C syntax which you are already used to will be identical to the specification syntax used in the specification file.

More than one structure can be included in a specification file. Multiple specifications per file are allowed. In addition to allowing structure specifications, typedefs, enumerations, and constants can also be specified.

These are detailed below.

C.5.1. Definitions

A specification file syntactically consists of a series of *definitions*. Each definition should be separated by a semi-colon:

```
definition-list:  
    definition ;  
    definition ;  
    definition-list
```

There are five types of definitions allowed :

- Structures**
- Typedefs**
- Symbolic Constants**
- Enumerations**

There are no order dependencies in the specification file: structures, typedefs, constants, and enumerations can appear in any order. Types or variables should generally be defined before they are used.

C.5.1.1. Structures

Structures are declared as they are in C:

```
struct-definition:  
    struct struct-ident {  
        declaration-list  
    }  
  
declaration-list  
    declaration ;  
    declaration ; declaration-list
```

For example, a structure for an RGB color would be declared in the specification file as follows :

```
struct color {  
    float red;  
    float green;  
    float blue;  
};
```

This would generate the following in the header file :

```
struct color {  
    float red;  
    float green;
```

```
float blue;
};
extern int KTYPE_COLOR;
int kdefine_color_PROTO((void));
```

For this example, the generated structure corresponds exactly with the specified structure. The integer `KTYPE_COLOR` is a type identifier which is initialized by the function `kdefine_color()`.

C.5.1.2. Typedefs

Typedefs are also declared as they are in C:

```
typedef declaration;
```

Typedefs generate a type identifier and also must be defined. Note that in defining the typedef of a structure, the structure will also be defined. Thus it is only necessary to call the define routine for the typedef.

For example, the following typedef defines a color type corresponding to the color structure specified in earlier :

```
typedef color color_t;
```

This would generate the following in the header file :

```
typedef color color_t;
extern int KTYPE_COLOR_T;
int kdefine_color_t_PROTO((void));
```

For this example, calling the `kdefine_color_t()` call would initialize the `KTYPE_COLOR_T` and `KTYPE_COLOR` identifiers. From then on, either identifier could be used when dealing with color structures.

Note that in addition to structures, any regular C data type can be typedef'ed. In all cases, a define routine will be created and must be called before the type identifier can be used.

C.5.1.3. Symbolic Constants

The declaration of symbolic constants is also similar to the regular C syntax:

```
const const-ident = integer
```

The generated constant produces a corresponding `#define` in the generated header file.

For example, the following symbolic constant in the specification defines a maximum size to be used, perhaps, in declaring arrays.

```
const MAX_SIZE = 1024;
```

This would generate the following in the header file :

```
#define MAX_SIZE 1024
```

This `#define` is available for use in any declarations subsequent to the constant declaration. This is handled by the C preprocessor when the code which includes the header file is compiled.

C.5.1.4. Enumerations

Enumeration declarations are also similar to C :

```
enum-definition:
    enum enum-ident {
        enum-value-list
    }

enum-value-list:
    enum-value
    enum-value , enum-value-list

enum-value:
    enum-value-ident
    enum-value-ident = value
```

For example, the following enumeration might be declared to define interpolation types :

```
enum interpolation {
    NONE = 0,
    ZERO_ORDER = 1,
    FIRST_ORDER = 2
};
```

This would generate the following in the header file :

```
enum interpolation {
    NONE = 0,
    ZERO_ORDER = 1,
    FIRST_ORDER = 2,
};
typedef enum interpolation interpolation;
extern int KTYPE_INTERPOLATION;
int kdefine_interpolation PROTO((void));
```

Notice that enumerations get typedef'ed to allow you to use them as a data type. A corresponding `KTYPE_INTERPOLATION` is also generated along with a define routine allowing you to define the enumerated type as a new VisiQuest data type.

C.5.2. Declarations

Declarations are the basic component of a specification. For instance, structures are composed of component fields, each field consisting of a variable declaration. Even typedefs define a type name for a declaration. For the most part, these declarations are similar to C.

There are four basic declarations allowed:

- **Simple**
- **Fixed Array**
- **Variable Array**
- **Pointer**

With the exception of variable arrays, these are all identical to C.

C.5.2.1. Simple Declaration

A *simple declaration* is a type identifier, followed by a variable identifier. These are propagated unchanged into the header file.

For example, the simple declaration :

```
int foobar;
```

will appear identical in the header file.

C.5.2.2. Fixed Array Declaration

A *fixed array declaration* is simply a type identifier, followed by an variable identifier with the array size in brackets. These are propagated unchanged to the header file.

For example, the simple declaration :

```
int choices[9];
```

will appear identical in the header file.

C.5.2.3. Variable Array

Since C contains no syntax to denote *variable-sized arrays*, a special syntax was created to support this. The declaration is similar to fixed array declarations, only that angle brackets are used instead of square brackets. If no number is specified between the angle brackets, the variable array can be any size. If a number is specified, then the variable array is limited to that size.

For example, the following two declarations illustrate two variable array declarations of type integer.

```
int x<MAX>; /* at most MAX elements */
int y<>;    /* any number of elements */
```

From this specification, a structure which reflects the size of the allocated variable length array is generated. This structure contains a `_len` field for storing the size and a `_val` field which points to the allocated array. These component names will begin with the name of the declared variable.

Thus, for the above example specification, the following will be generated in the header file :

```
struct {
    unsigned int x_len;
    int *x_val;
} x;
struct {
    unsigned int y_len;
    int *y_val;
} y;
```

It is up to the programmer to manage allocation of the `_val` field and to maintain the `_len` field so it correctly reflects the size of the variable length array. Maintaining the `_len` field is important, as the generated structure code uses that field to know how many elements are in the array, and thus, how many elements should be written when the structure is stored.

C.5.2.4. Pointer Declaration

A *pointer declaration* consists of a type identifier, followed by an asterisk, followed by a variable identifier. These are propagated unchanged into the header file.

For example, the pointer declaration :

```
int *next;
```

will appear identical in the header file.

Note that the pointers themselves will not be written and read, as they are just memory addresses. The pointer is followed, with the single item being pointed being read and written.

Pointers are tracked for any given structure written out, so that multiple pointers to a single item are not followed multiple times. These multiple pointers will be reconnected accordingly when the structure is read back in.

C.5.2.5. Type Identifiers

The *type identifiers* used in any of these declarations can be any of the standard C data types, from unsigned char to int to double. Additionally, they can be of any declared structure, typedef, or enumerated type.

Strings are treated as a special case, with a string being treated as a variable type identifier rather than an array of bytes. The specific type identifier which should be used for string variables in the specification file is `kstring`.

Internally, `kstring` is simply a typedef to `char *`. Note however that you must explicitly use `kstring` instead of `char *`, as it will indicate to `kgenstruct` that the generated code needs to specifically handle strings.

C.5.3. Structure Versioning

Structures often evolve and change in a typical software development process. This becomes critical when reading and writing structures since, if a structure changed, previously written instances of that structure can no longer be read. Because of this problem, support for adding a version number to each structure has been

created. While structures version numbers are optional, their use provides the ability for recognizing mismatches between stored structures and defined structures. It provides a mechanism for the structure reading code to recognize out-of-date structures. Without the versioning, the structure reader for the new structure would try, probably unsuccessfully, to read the stored instance of the old structure.

A version number is specified after the structure name with the word "version" and a number. The following example illustrates a 2.0 version of the structure "foobar".

```
struct foobar
  version 2.0
{
  int a;
  int b;
};
```

Assuming a new field was added to this structure, it would be appropriate to change the version number :

```
struct foobar
  version 2.1
{
  int a;
  int b;
  int c;
};
```

The reading of a stored structure that is out of date with the currently defined version will fail. Note that a versioned structure will be considered different from a non-versioned structure, even though the actual structures may be identical.

Typedefs of structures, when included in the same specification file as the structure being typedef'ed, will inherit the version number of the structure. As with structures, a versioned typedef is considered different from a non-versioned typedef.

C.5.4. C Preprocessing

Kgenstruct passes each specification file through the C preprocessor before parsing it to generate code. This allows you to use CPP directives within your .x file. For example, `#include` can be used to include other specification files, or even header files, and `#define` can be used to define special symbols needed only in the specification file.

C.5.4.1. Comments

C-style comments, beginning with `/*` and ending with `*/`, are allowed within the specification and will be ignored. For example, the following line would be ignored.

```
/* This line is considered a comment and will be ignored */
```


C.5.4.2. Passing Directives to Header File

Lines beginning with "%" are passed through to the header file, without the "%". This allows you to customize your generated header file with specific variables or C preprocessor directives. One potential use of this construct would be to pass directives to extern structures for use with C++. The following example externs the foobar structure.

```
%#ifndef _cplusplus
%extern "C" {
%endif
struct foobar {int a; int b};
%#ifndef _cplusplus
%}
%endif
```

C.5.5. Additional Notes

The program definition supported by rpcgen will be recognized by kgenstruct, but is not explicitly supported and should be used at your own risk. The union construct is also not supported. The union construct, as used in rpcgen, is not identical to the C union, but is instead used to handle conditional reading and writing.

C.5.6. Example Specification File

The following is an example of a specification file which illustrates many of the details explained earlier.

```
/*
**** example.x file
****
**** This file defines a bogus hypothesis and tile structure
**** for the purposes of illustrating a kgenstruct. Note that
**** the specification is practically identical to C.
****/

/* Note also that C comments are allowed in the specification. */

**** Anything following a '%' will be passed through (without
**** the '%') to the generated header file io_example.h.
**** This is useful if you need to include any CPP directives
**** of your own.
****/

**** For instance, the following line will generate a #include
**** in the .h file
****/
#include "my_defines.h"

****
**** HYPOTHESIS structure
****/
struct hypo
{
    int    position;
    float  probabilities[10];

    float  uncertainty<>;
```

```

    struct hypo *next;
};

    /****
    ****  HYPOTHESIS typedef
    ****/
typedef struct hypo hypo_t;

    /****
    ****  TILE structure
    ****/
struct tile
    version 1.0
{
    int id;
    hypo_t hypos[10];
};

    /****
    ****  TILE typedef
    ****/
typedef struct tile tile_t;

```

D. Data Services

Data Services provides access to an abstract data object. Typical access to a data object is in terms of one of the defined data models, for example the polymorphic data model or the geometry data model. Each of these data models, implemented in terms of an application service, is built on a basic infrastructure for storing data segments and attributes. The ability to create segments and attributes is available to any programmer. In particular, the ability to create new attributes is *encouraged* as method of extending the existing data models to support your specific data needs.

This capability is greatly strengthened by the ability to understand defined structures. Once a structure is defined as a new data type, it is possible to *store and retrieve instances of that structure from attributes in a data object*. An attribute of a structure data type can be defined or created, and then accessed using the standard `get_attribute` and `set_attribute` calls available with Data Services.

D.1. Attributes Characteristics

Attributes have a number of characteristics which define their behavior, but they are fundamentally distinguished by their name. Each attribute name is a unique string identifier, no two attributes can have the same name. This name is typically `#defined` to some unique identifier.

The *number of arguments* in an attribute indicate whether the attribute consists of one or more arguments. These arguments will correspond to multiple values or variables when being accessed, with each argument being comma separated.

The *argument size* of each argument dictates whether each attribute argument is a single value, or an array of values. Each argument will take on this characteristic.

The *data type* of an attribute dictates what data type each argument element will be. Each argument takes on this data type. If an attribute contains an argument size greater than one, each argument will be an array of

elements of this data type. The type identifier for a defined structure can be used to dictate the data type of an attribute. It is recommended for structure attributes that the type identifier for the structure *and not a type identifier for a typedef of the structure* is used to specify the data type.

The *permanence* of an attribute dictates whether the attribute should be stored to output when the object is closed or not.

An attribute is considered to be *shared* if it exists at the physical level of a data object. This is useful if you are working with reference objects. Data objects are divided into a physical level and a presentation level. Each reference of a single data object will consist of unique presentation layer on top of a common physical layer. Attributes at the physical layer are thus shared among all reference objects. Attributes at the presentation layer are unique to each reference object and are thus not shared.

Note also that attributes are created within a data object at a specified scope. This is often termed to be the *association* of the attribute. Attributes can be associated with either a specific data segment, or with the overall data object. Attributes are most often associated with the data object, signified by a NULL association.

D.2. Attributes Management

Attributes can be instantiated in two different ways. The most direct way is to *create* the attribute. Create routines exist within all the existing application services which allow you to create an attribute. The created attribute exists only for the specific object in which it was created. The other way to instantiate an attribute is to *define* the attribute. Once defined, an attribute will appear to exist over all objects. If a defined attribute is retrieved from a data object which does not actually contain an instance of the attribute, an instance is created before the attribute is accessed. This makes adding a set of attributes which will be widely used quite easy. Note that the define attribute capability is only available with the `kdms_define_attribute()` call.

There are six specific functions for working with attributes which have either been defined or created. Versions of these calls exists for each of the application Data Services.

Get Attribute : assigns an attribute to a data object

Set Attribute : retrieves an attribute from a data object

Query Attribute : indicates if an attribute exists and returns its characteristics if it does

Copy Attribute : copies attribute values from one object to another

Match Attribute : compares attribute values across two objects

Print Attribute : prints attribute values

All these work as expected for structure attributes, with the exception of the print function, which will only print the data type of the structure attribute.

D.3. Handling Undefined or Mismatched Versioned Structures

Data Services will only be able to read an structure attribute if the structure type identifier has been defined before the data object is opened. In addition to being defined, the structure stored in the object must have the

same version as the structure which has been defined. If the structure type identifier has not been defined, or there is a version mismatch, Data Services will fail to read the attribute when the object is opened. This condition will generate an error message for you indicating the name of the attribute, and its type and version.

Beyond this, the data object will still be opened successfully, with all other attributes and data being available within the program. The specific structure attribute which failed to read in, however, will not be instantiated in the data object. The programmer can detect the absence of this attribute in one of two ways, depending whether the attribute was created or defined.

If the attribute was defined, then a default value specified in the define attribute call will be returned when the attribute is retrieved from the object. By defining the default value to be NULL, a programmer can then tell, on retrieval of a NULL value, that the input object either did not contain the structure, or contained a mismatched version of the structure.

If the attribute is not defined, but instead was created in the previous routine, then it will simply just not exist in the newly opened data object. Calls to retrieve the attribute will fail and query attribute calls will indicate that the attribute does not exist.

Note that defined attributes appear to always exist, whether or not they are really physically instantiated on the data object; a query attribute will always indicate that a defined attribute exists on any data object.

D.4. Nuances of Attribute Assignment

In general, attributes with an argument size of one are stored and retrieved by value. That is, a copy of the value is made to internal storage when the attribute is set. A copy of the value in internal storage is made back to the retrieving variable when the attribute is returned. This is true for all C data types.

The following example illustrates setting and getting a simple integer attribute of argument size one.

```
int set_value = 10;
int get_value;

kpds_set_attribute(object, ATTRIBUTE, set_value);
kpds_get_attribute(object, ATTRIBUTE, &get_value);
```

Attributes with an argument size greater than one are treated differently. Since each attribute argument is really an array, there is a greater overhead involved in copying the attribute argument than a simple C assignment. This copying cost is incurred on a set attribute. This is done to protect the data integrity; once set, the data object has its own private copy of the attribute argument which you can not accidentally free. To avoid some of the overhead of copying, returning attributes are passed by reference. That is, only a pointer to the internal array is returned.

The following example illustrates setting and getting a simple integer attribute of argument size three.

```
int set_value[3] = {4, 5, 6};
int *get_value;

/* this will copy the set_value array to an internal array */
kpds_set_attribute(object, ATTRIBUTE, set_value);

/* this will return the pointer to an internal array */
kpds_get_attribute(object, ATTRIBUTE, &get_value);
```

Structures are treated as a special case. Since allocated structures can not be passed on the stack, it is necessary to pass the address of the structure, or a pointer to the structure. Structure arguments are handled in a manner similar to array arguments. An internal copy of a structure will be made on a set attribute call and a pointer to that internal array will be returned on a get attribute call.

The following example illustrates setting and getting a simple structure attribute with an argument size of one.

```

struct tile *set_tile;
struct tile *get_tile;

set_tile = (struct tile *) allocate_and_initialize_tile();

/* this will create an internal copy of the set_tile structure */
kpds_set_attribute(object, ATTRIBUTE, set_tile);

/* we can free this copy */
kfree(set_tile);

/* this will return the pointer to an internal structure */
kpds_get_attribute(object, ATTRIBUTE, &get_tile);

/* don't free get_tile! */

```

D.5. Structure Storage within a Data Object

The actual unflattening and flattening of a structure attribute occurs when a data object is opened and closed. When an input object is opened, any stored structure attributes which are defined are unflattened from the file into memory. When an output object is closed, any structure attributes in that object are flattened from memory into the file. Note that any internal copies of the structure within the data object will be freed when the data object is closed.

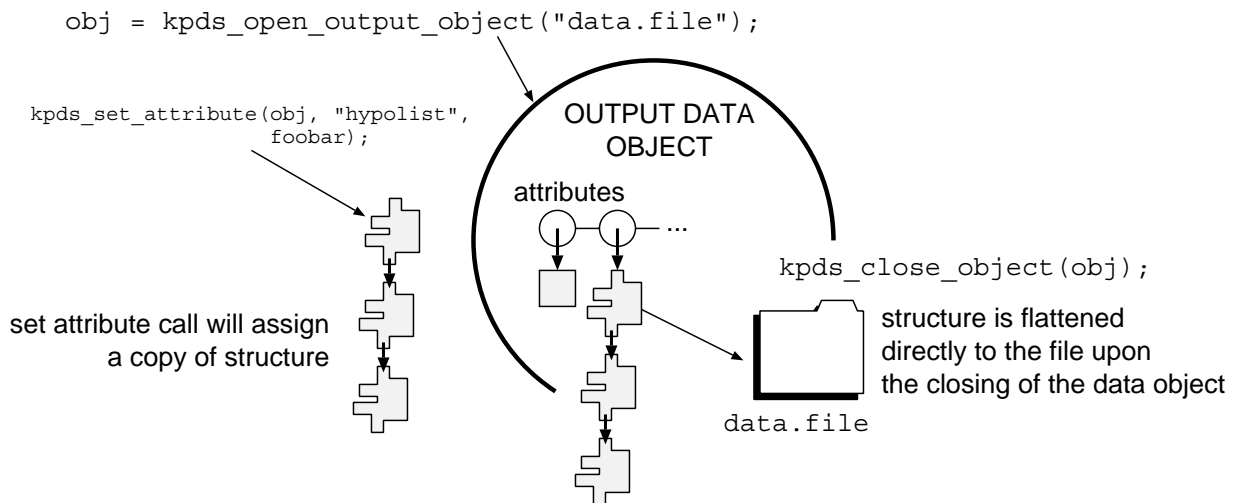


Figure 3: This picture illustrates the flattening of a structure attribute from memory to a file as a data object is closed. Notice that when the attribute was stored, an internal copy of the structure was made.

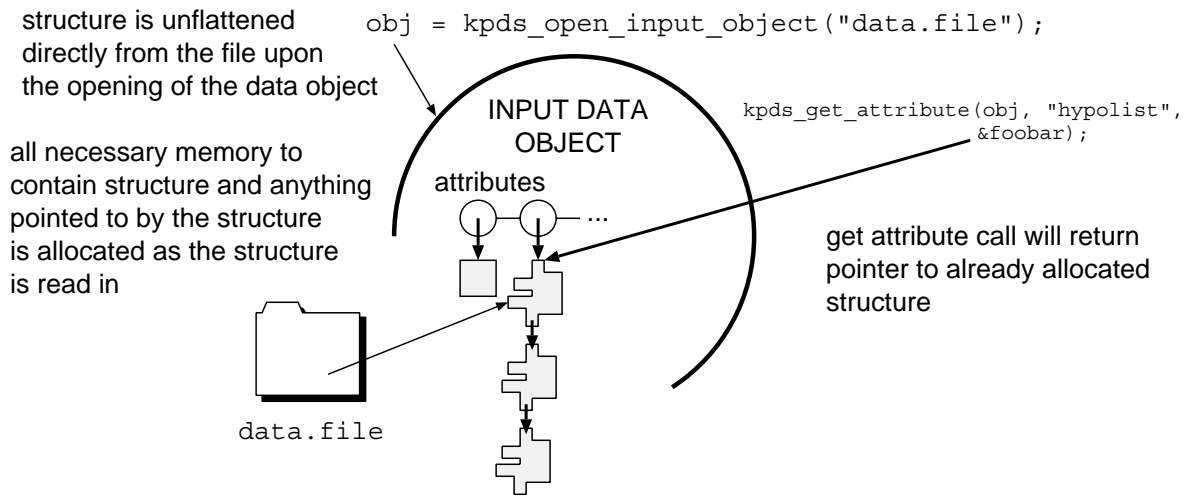


Figure 4: This picture illustrates the unflattening of a structure attribute from memory as a data object is opened. Notice that when the attribute is retrieved, only a pointer to the already allocated structure is returned.

The code dealing with data storage for a data object will store a machine identifier, using it appropriately so that any flattened structures on one architecture can be unflattened appropriately on another. This machine conversion will be entirely transparent to you when using Data Services.

Should you need to handle writing out flattened and unflattened versions of a structure yourself, there are calls for performing the flattening and unflattening entirely in memory. These calls, `kstruct_flatten` and `kstruct_unflatten`, will perform the flattening in the native machine format, so be careful to manage your own machine conversion information when transporting the flattened structures across machines.

E. Tutorial : Creating a Structure Support Library

This tutorial will illustrate the creation of a structure specification and some corresponding library code for initializing the structures. This tutorial is written assuming a library object named `ATRDATA` has been created and is being edited with Composer.

1. The first step is to create a structure specification file. We will create three structures within this file. A specification file is created by adding a new *SRC* file in Composer using the "Add File to Object" operation under the "File Operations" pulldown menu. Using this operation, add a file named **hypo.x**.
2. The next step is to edit the specification file and add the structure specification. Using the "Edit" operation under the "File Operations" pulldown menu, edit the file **hypo.x** and add the following structure specification :

```
struct hypo
  version 1.0
  {
    float height;
```

```

        int   type;
        float peaks<>;
    };

    struct hypo_list
        version 1.0
    {
        struct hypo h;
        struct hypo *next;
    };

    struct tile
        version 1.0
    {
        int x;
        int y;
    }

    typedef struct hypo_list hypo_list_t;
    typedef struct tile tile_t;

```

- Now that we have a specification file, generate code from it using the *Make* button found on the *on Console* option from the *Commands* pulldown menu. This will invoke the program *kgenstruct* to parse the specification file **hypo.x** and generate two code files *io_hypo.h* and *io_hypo.c*.
- The generated header file should be included by the library's include file so that it is available publicly. Edit the library's include file, *atrdata.h* and add the following line in the *#include* section.

```
#include "io_hypo.h"
```

- We will want to define two data services attributes. For convenience, we will define two strings in the *atrdata.h* file. Edit the library's include file, *atrdata.h* and add the following line in the *#define* section.

```
#define ATR_HYPO_LIST "atrHypoList"
#define ATR_TILE      "atrTile"
```

- Now, we need to create an initialization routine which will define the structures and define some Data Services attributes. Create a new file called **init_hypo.c** and add in the following procedure. This procedure will define the structures, and define a specific attributes for transporting them via Data Services.

```

/*-----
| Routine Name: init_structures
|
| Purpose: This routine will initialize the data types
|         tile_t and hypo_list_t for use with data
|         services in the rest of the program.
|
|         The attributes ATR_HYPO_LIST and ATR_TILE
|         will also be defined here.
|
|         This routine should be called once as an
|         initialization step for the rest of the program.
|
|-----

```

```

|           The type handlers for these will be
|           KTYPE_TILE and KTYPE_HYPO_LIST.
-----*/
void init_structures(void)
{
    static int initialized = FALSE;

    if (!initialized) /*-- only initialize once --*/
    {
        initialized = TRUE;

        /****
        **** these define calls will recursively define the structures
        **** hypo_list, hypo, and tile.
        ****/
        kdefine_hypo_list_t();
        kdefine_tile_t();

        /****
        **** define the two data services attributes. These
        **** will each be single argument attributes with an
        **** argument size of 1. We will associate it at the
        **** object level and provide NULL as a default.
        ****/

        kdms_define_attribute(NULL, ATR_HYPO_LIST, 1, 1, KTYPE_HYPO_LIST,
                               TRUE, TRUE, NULL);

        kdms_define_attribute(NULL, ATR_TILE, 1, 1, KTYPE_TILE,
                               TRUE, TRUE, NULL);
    }
    return;
}

```

7. Be sure to properly prototype this generated routine by adding the following line to the **atrdata.h** library include file in the routine definitions section.

```
void init_structures PROTO((void));
```

8. You can now compile the library. The routine `init_structures()` can now be used by any routine which wants to have access to the the structure attributes `ATR_HYPO_LIST` and `ATR_TILE`. Note that this initialization routine should be called **first**, before any other calls to data services. If, for example, an object containing a `hypo_list` were opened before this routine was called and the `KTYPE_HYPO_LIST` was defined, the `hypo_list` could not be read.
9. In a program linked against this library, you can access the `ATR_HYPO_LIST` and `ATR_TILE` attributes transparently once you have called the `init_structures()` routine.

To write a hypo list on a Data Services object, simply use a set attribute call as shown in the following example. The declaration of the hypo list structure is shown as well, although presumably the hypo list would be initialized to contain some useful information before the set attribute call.

```

kobject          output_obj;
struct hypo_list_t *hypo_l;

output_obj = kpds_open_output_object("filename");

```



```
kpds_set_attribute(output_obj, ATR_HYPO_LIST, hypo_l);
kpds_close_object(output_obj);
```

Note that the hypo will not be written to *filename* until the `kpds_close_object()` call.

10. The hypo list is read from a Data Services object in a similar manner. To retrieve a hypo list from a Data Services object, simply use a get attribute call as shown in the following example. The declaration of the hypo list structure is also shown.

```
kobject          output_obj;
struct hypo_list_t *hypo_l = NULL ;

output_obj = kpds_open_input_object("filename");
kpds_get_attribute(output_obj, ATR_HYPO_LIST, &hypo_l);
kpds_close_object(output_obj);
```

Remember that this returns a pointer to the internal Data Services copy of the structure. As such, it will be freed during `kpds_close_object()` call. The routine `kstruct_duplicate()` can be used to make a copy of a structure, if a private copy which Data Services won't free is needed.

This page left intentionally blank

Table of Contents

A. Overview	6-1
B. Passing Structures Between Programs	6-2
C. Structure Specification File	6-3
C.1. Creating a Structure Specification File	6-3
C.2. Kgenstruct	6-3
C.3. What is Generated?	6-3
C.4. Software Object Types	6-4
C.5. Specification Language	6-4
C.5.1. Definitions	6-5
C.5.1.1. Structures	6-5
C.5.1.2. Typedefs	6-6
C.5.1.3. Symbolic Constants	6-6
C.5.1.4. Enumerations	6-7
C.5.2. Declarations	6-7
C.5.2.1. Simple Declaration	6-8
C.5.2.2. Fixed Array Declaration	6-8
C.5.2.3. Variable Array	6-8
C.5.2.4. Pointer Declaration	6-9
C.5.2.5. Type Identifiers	6-9
C.5.3. Structure Versioning	6-9
C.5.4. C Preprocessing	6-10
C.5.4.1. Comments	6-10
C.5.4.2. Passing Directives to Header File	6-11
C.5.5. Additional Notes	6-11
C.5.6. Example Specification File	6-11
D. Data Services	6-12
D.1. Attributes Characteristics	6-12
D.2. Attributes Management	6-13
D.3. Handling Undefined or Mismatched Versioned Structures	6-13
D.4. Nuances of Attribute Assignment	6-14
D.5. Structure Storage within a Data Object	6-15
E. Tutorial : Creating a Structure Support Library	6-16

This page left intentionally blank

Chapter 7

Streaming Data Services

Chapter 7 - Streaming Data Services

A. Introduction

Streaming Data Services provides a mechanism for the continuous transmission of data messages from one data processing routine to another. Streaming Data Services was designed to have minimal overhead so that it could be used for processing of data in real-time. Streaming Data Services allows multiple "channels" of data to be multiplexed down a single data stream. The data being streamed down a channel can be of any VisiQuest supported datatype. Arbitrary user-defined data structures can even be streamed.

Streaming Data Services provides access to a stream connection via an opaque data object represented by the abstract data type *kobject*. A single stream connects two data processing operators which run simultaneously. One process will continuously produce data which the other will continuously consume.

A data stream is either opened as a *sending* stream for writing or as an *receiving* stream for reading. A sending data stream should be opened by one process for communication with a second process. The second process should then open the *same* stream as a receiving stream. The first process then writes data to its sending stream and the second process reads the same data from its receiving stream.

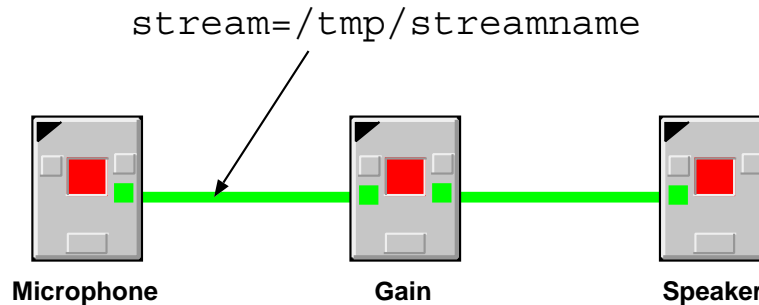


Figure 1: A data stream is used to continuously transmit data between two simultaneously running processes. The stream is identified by a unique transport name. In VisiQuest, the transport type used to transmit the stream can be changed by clicking on the connection between two glyphs.

Data can be written *only* to a sending stream and can be read *only* from a receiving stream. Communication down a stream is thus unidirectional. The second process can not communicate back to the first process using the same stream; a second stream must be created.

A VisiQuest transport name is used to identify the stream name. The sending filename should begin with the prefix *stream=* to truly be a stream; the prefix should *not* be specified by the receiving process. Other transient transport types, such as a UNIX-domain or TCP/IP sockets may also be used. Streaming data services will automatically handle any machine conversion required when transmitting data between different architectures. Other permanent transport types such as file or shared memory may be used, although this will cause all the data written to the stream to be buffered into the transport.

Note: When editing a kroutine from guise, use `stream=` as the default file name for all streaming *sending* connections. This will cause the connection type to automatically be a stream. The data being sent and received consists of basic one-dimensional arrays. No specific structure is imposed on the data; it is up to the application to enforce a more specific interpretation. To help with this, streaming data services supports user-defined attributes which allow application-specific information to be passed down a data stream.

For example, while the data composing a two-dimensional image could be sent down a data stream, the downstream process would not be able to reconstruct the image without more specific information such as the image width and height. Thus, an user could define attributes which describes the image width and height.

B. Streaming Data Model

Streaming Data Services provides both a writing and a reading process access to a stream. The stream allows data to be sent from the writing process to the reading process. A writing process will communicate with a reading process by sending data through a stream using the `kds_write_data` call. The reading process receives this data using the `kds_read_data` call.

The data being sent with a `kds_write_data` call consists of a single linear array of data. This array is written to the data stream where it can then be read by the reading process. Thus, even though the data is conceptually streaming continuously from one process to another, it is in reality transferred one array at a time. For convenience, this array will often be referred to as *a message*. Note that streaming data services does no buffering of data. Subsequent sends will block until the message from a previous send has been completely received by the downstream process.

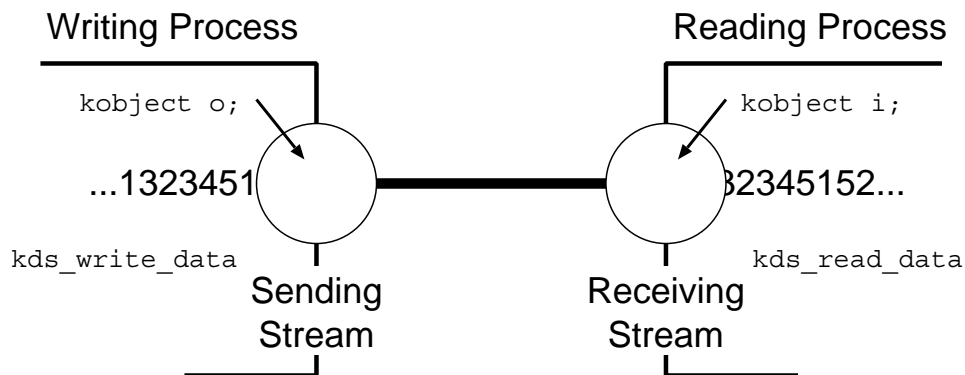


Figure 2: Data is written do a data stream using the `kds_write_data` call. That same data is read from the stream in another process using the `kds_read_data` call. Each send consists of a linear array of data called a *message*. Data can flow in only one direction down a stream.

A stream may contain multiple data channels which act as conduits for sending and receiving messages. Channels may also act as conduits for additional, auxiliary information known as attributes.

Streams

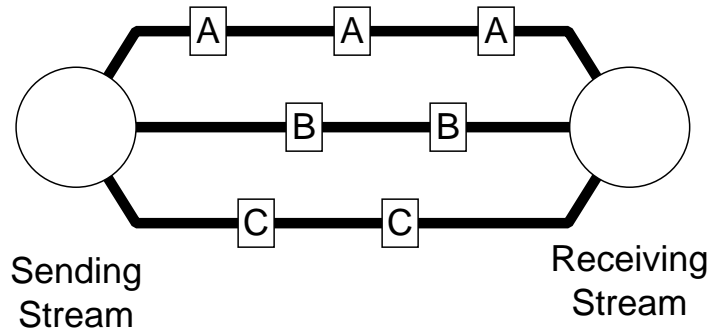
A stream is represented by the abstract datatype `kobject`. A stream should either be opened as a sending stream with the `kds_open_send` call, or as a receiving stream with the `kds_open_recv` call. The single argument to either of these functions will be the transport name over which the stream will be communicated; typically this should be prefixed with

`stream=`. These calls will return a valid `kobject` on success or `NULL` on failure.

Channels

While multiple channels can be opened within a stream, data cannot be sent simultaneously down multiple channels at the same time. Data down multiple channels is actually interleaved; each subsequent `kds_write_data` call may be sent down a different channel on the same stream.

For example, in the illustration below, three channels named *A*, *B*, and *C* are opened in the output stream. The output stream first does a `kds_write_data` call with a message down channel *A*. The next send is down channel *B*, the next down channel *C*, and so forth. Notice that the sending stream is driving what is being received. Since no data is buffered, the receiving stream must receive the messages *in the order in which they have been sent*.



Channels are opened only on sending streams. Once a channel is opened, attributes may be set on the channel, and then the channel should be connected. Once the channel is connected, it will automatically be opened and connected in the downstream input object. This will be discussed in more detail later.

Attributes

In addition to data, a channel may also act as a conduit for attributes. An attribute is a named piece of information with a specific value. An attribute value may be assigned with a call to `kds_set_attribute` or its value retrieved with a call to `kds_get_attribute`. User-defined attributes must be created explicitly on a channel with a call to `kds_create_attribute` before it can be accessed.

Attributes for Streaming Data Services are used to define a process's view of a channel rather than being associated with the data itself; it is possible for every process to have a different view (set of attributes) associated with a channel and its data stream. A process may assign an attribute through functions such as `kds_set_attribute`, but this attribute only exists locally. The functions `kds_connect_channel` and `kds_resend_attributes` can be used to propagate local attributes to the next downstream glyph which then become part of that glyph's local attributes. Attributes cannot be propagated upstream.

Attributes typically should be created and set on a channel after it is opened but before it is connected. The call to connect a channel sends all the attributes of the channel downstream. If created before the `kds_connect_channel` call (or resent with `kds_resend_attributes`) then attributes need only be created on sending streams; the attributes will be created automatically on the downstream receiving stream when they are received.

Each channel has one attribute which **must** be set. The attribute `KDS_DATATYPE` dictates the datatype of the messages sent down the channel. Once set, this attribute applies for the life of the channel; it cannot be changed, even if the attributes are resent. Any of the VisiQuest standard datatypes listed on the left may be used when setting this attribute. User-defined datatypes may be used as well if streaming of arbitrary structures is required. Note that the `KDS_DATATYPE` attribute exists inherently in each channel and does not need to be created.

```
KBIT
KBYTE
KUBYTE
KSHORT
KUSHORT
KINT
KUINT
KLONG
KULONG
KFLOAT
KDOUBLE
KCOMPLEX
KDComplex
```

Attributes generally have to be explicitly created on a sending channel before they can be set. However, this can be avoided with attributes which are used frequently by defining them using the `kds_define_attribute` call. A defined attribute will be automatically created on a call to `kds_set_attribute`, thus saving the explicit call to create the attribute. It is recommended that common attribute definitions be placed in a single library to be shared among the multiple streaming routines who need them. Not only will this avoid the extra step of having to create the attribute before setting it, but it also guarantees that the attribute will be created with the same characteristics within each routine.

C. Stream Functions

Depending on whether or not you are working with just a sending stream, just a receiving stream, or both a sending and receiving stream, the function calls which should be used and the general programming model will vary. The general functions for sending and receiving streams are outlined below, with examples of how each processing case should look provided at the end.

C.1. Sending Stream Functions

Data is communicated down a sending stream via channels. Channels are identified by a unique character string. This string is provided when the channel is opened with a `kds_open_channel` call. At this time, the `KDS_DATATYPE` attribute be set on the channel. This attribute will determine the datatype of the messages sent with a `kds_write_data` call. All data sent over this channel *must* be of this datatype. This datatype will hold for the life of the channel. If different datatypes are to be sent, then `KBYTE` can be used for the datatype, but then the receiving channel must know how to interpret the data and there will not be automatic conversion of datatypes between heterogeneous machines.

Other attributes may be created and set as needed after the channel has been opened. Once all attributes are set on the channel, it should then be connected using the `kds_connect_channel` call. Once connected, the datatype and all other channel attributes will be sent downstream.

Channels should only be opened and connected on sending streams. Once a channel has been connected, it will be automatically opened and connected within the downstream receiver. A channel may be closed on an sending stream at any time using the `kds_close_channel` call. This will automatically close the channel downstream as well. All channels are closed automatically when a stream is closed with the `kds_close` call. Channels on receiving streams should never be explicitly closed.

Note that the channel name will be internally bound to a unique **byte** identifier within a stream. This byte is sent at the beginning of each message to identify to which channel the message belongs. While this minimizes the overhead in sending each message, it does impose a practical limit of 255 channels in each data stream.

- `kds_open_send()` - open a sending stream
- `kds_open_channel()` - open a sending channel
- `kds_connect_channel()` - connect an sending channel
- `kds_write_data()` - write data down a sending channel
- `kds_close_channel()` - close a sending channel
- `kds_close()` - close a stream

C.1.1. `kds_open_send()` — *open a sending stream*

Synopsis

```
kobject  
kds_open_send(kstring name)
```

Input Arguments

```
name  
    name of stream
```

Returns

```
kobject on success, NULL otherwise
```

Description

```
This function will open a stream for writing.
```

C.1.2. `kds_open_channel()` — *open a sending channel*

Synopsis

```
int
kds_open_channel(
    kobject object,
    kstring channel)
```

Input Arguments

```
object
    stream object to contain channel
channel
    name of new channel to open
```

Returns

TRUE on success, FALSE otherwise

Description

This function will open a channel within the given data stream. After the channel is opened, attributes may be set using the `kds_set_attribute` function. When all attributes have been set, the channel may be connected using the `kds_connect_channel` function.

Only channels on a sending stream should be opened. Receiving channels will be implicitly opened within the `select` or the `kds_read_data` call.

C.1.3. `kds_connect_channel()` — *connect an sending channel*

Synopsis

```
int
kds_connect_channel(
    kobject object,
    kstring channel)
```

Input Arguments

```
object
    stream object to contain channel
channel
    name of new channel to connect
```

Returns

TRUE on success, FALSE otherwise

Description

This function will connect an open channel within the given data stream. After the channel is connected, the datatype attribute may no longer be changed. The connect effectively transmits all attributes which were set after the channel was opened. The KDS_DATATYPE attribute must have been set before this call is made.

Only channels on sending streams should be opened. Receiving channels will be implicitly opened within the select or the kds_read_data call.

C.1.4. kds_write_data() — *write data down a sending channel*

Synopsis

```
ssize_t
kds_write_data(
    kobject object,
    kstring channel,
    kaddr data,
    size_t number)
```

Input Arguments

```
object
    sending stream object
channel
    channel associated with data
data
    write data buffer
number
    number of items in data buffer
```

Returns

number of items successfully sent or -1 on error

Description

Writes data to a channel on a sending stream. The datatype of the data to be transmitted down this channel must have been set prior to the channel being connected.

C.1.5. `kds_close_channel()` — *close a sending channel*

Synopsis

```
int
kds_close_channel(
    kobject object,
    kstring channel)
```

Input Arguments

```
object
    stream object to contain channel
channel
    channel being closed
```

Returns

TRUE on success, FALSE otherwise

Description

Close a sending channel, indicating that no further data will be written to it. This will signal to the downstream process that this channel is being closed.

Only sending channels should be closed. Receiving channels will be implicitly closed within the select or the `kds_read_data` call.

C.1.6. `kds_close()` — *close a stream*

Synopsis

```
int
kds_close(kobject object)
```

Input Arguments

```
object
    stream object to close
```

Returns

TRUE on success, FALSE otherwise

Description

This function will close a stream.

C.2. Receiving Stream Functions

The order in which the messages arrive at the receiving stream is determined entirely by the order in which they were sent from the upstream process. Since messages must be received in the order in which they were sent, the reading process must receive each message as it arrives.

The `kds_select_channel` call should be used on an receiving stream to determine the channel on which the next incoming message is arriving. The `kds_select_channel` call will intercept each new incoming message as it arrives and then indicate to the calling program to which channel it belongs. *The calling program must receive the data from the selected channel before any subsequent messages can be received.*

- `kds_open_recv()` - open an receiving stream
- `kds_select_channel()` - select next receiving channel with available data
- `kds_read_data()` - read data from a receiving channel

C.2.1. `kds_open_recv()` — *open an receiving stream*

Synopsis

```
kobject  
kds_open_recv(kstring name)
```

Input Arguments

```
name  
name of stream
```

Returns

kobject on success, NULL otherwise

Description

This function will open a stream for reading.

C.2.2. `kds_select_channel()` — *select next receiving channel with available data*

Synopsis

```
kstring  
kds_select_channel(kobject object)
```

Input Arguments

```
object  
receiving stream object
```

Returns

channel - name of next channel to read from

Description

This function will select the next incoming channel with data available. A channel which has just connected or a channel which as resent attributes will also return from this function.

C.2.3. `kds_read_data()` — *read data from a receiving channel*

Synopsis

```
ssize_t  
kds_read_data(  
    kobject object,  
    kstring channel,  
    kaddr data,  
    size_t number)
```

Input Arguments

```
object  
receiving stream object  
channel  
channel associated with data  
data  
receive data buffer  
number  
the maximum number of elements of the channel datatype which can be read into the data buffer
```

Returns

number of items successfully received or -1 on error

Description

Reads data from a channel on a receiving stream.

C.3. Sending to Receiving Functions

These functions are used when relaying information from an incoming receiving stream to an outgoing sending stream.

- *kds_query_channel()* - query a channel
- *kds_relay_channel()* - relay a channel to a sending stream
- *kds_copy_channel()* - copy a channel and associated data

C.3.1. *kds_query_channel()* — *query a channel*

Synopsis

```
int
kds_query_channel(
    kobject object,
    kstring channel)
```

Input Arguments

```
object
    stream object to check
channel
    channel to check for
```

Returns

TRUE if channel is present, FALSE otherwise

Description

Check to see if a channel exists within a given data stream.

C.3.2. `kds_relay_channel()` — *relay a channel to a sending stream*

Synopsis

```
int
kds_relay_channel(
    kobject input,
    kstring chan_in,
    kobject output,
    kstring chan_out)
```

Input Arguments

```
input
    stream containing receiving channel
chan_in
    receiving channel
output
    stream containing sending channel
chan_out
    sending channel
```

Returns

TRUE if sending channel is successfully relayed, FALSE otherwise

Description

Connect an sending channel of the same datatype as the named receiving channel. This will only copy the datatype attribute from the receiving channel to the sending channel.

C.3.3. `kds_copy_channel()` — *copy a channel and associated data*

Synopsis

```
int
kds_copy_channel(
    kobject input,
    kstring chan_in,
    kobject output,
    kstring chan_out)
```

Input Arguments

```
input
    stream containing receiving channel
```

chan_in
receiving channel
output
stream containing sending channel
chan_out
sending channel

Returns

TRUE if channel is present, FALSE otherwise

Description

Copy the data for a channel coming in from a receiving channel and writing out to an sending channel. If the channel does not yet exist in the sending stream, it will be relayed from the receiving stream.

C.4. Attribute Functions

The following functions allow attributes to be defined or created on output streams, and then assigned and retrieved from a stream. Note that attributes should be created and assigned before a stream is connected. The act of connecting a stream will send all attributes in a channel downstream.

- *kds_set_attribute()* - set an attribute
- *kds_get_attribute()* - get an attribute
- *kds_create_attribute()* - create an attribute
- *kds_delete_attribute()* - delete an attribute
- *kds_delete_attributes()* - delete all attributes
- *kds_define_attribute()* - define an attribute
- *kds_resend_attributes()* - resend all attributes
- *kds_get_attribute_names()* - get attribute names

C.4.1. *kds_set_attribute()* — *set an attribute*

Synopsis

```
int  
kds_set_attribute(  
    kobject    object,  
    kstring    channel,  
    kstring    attribute,  
    kvalist)
```

Input Arguments

object
stream object containing channel
channel
name of channel associated with attribute
attribute
name of attribute to set

Returns

TRUE on success, FALSE otherwise

Description

This function will set an attribute associated with a channel which can later be written to a channel by `kds_connect_channel` or `kds_resend_attributes`.

C.4.2. `kds_get_attribute()` — *get an attribute*

Synopsis

```
int  
kds_get_attribute(  
    kobject    object,  
    kstring    channel,  
    kstring    attribute,  
    kvalist)
```

Input Arguments

object
stream object containing channel
channel
name of channel associated with attribute

Output Arguments

attribute
name of attribute to get

Returns

TRUE on success, FALSE otherwise

Description

This function will get an attribute either previously set by the same process or written to a channel by

an upstream process using `kds_connect_channel` or `kds_resend_attributes`.

C.4.3. `kds_create_attribute()` — *create an attribute*

Synopsis

```
int
kds_create_attribute(
    kobject object,
    kstring channel,
    kstring attribute,
    int datatype,
    size_t argsize,
    size_t numargs)
```

Input Arguments

`object`
stream object containing the channel

`channel`
name of channel associated with attribute

`attribute`
name of attribute to create

`datatype`
datatype of attribute

`argsize`
size of each attribute argument

`numargs`
number of attribute arguments

Returns

TRUE on success, FALSE otherwise

Description

This function will create an attribute associated with a channel in a specific stream object.

C.4.4. `kds_delete_attribute()` — *delete an attribute*

Synopsis

```
int
kds_delete_attribute(
```

```
kobject object,  
kstring channel,  
kstring attribute)
```

Input Arguments

```
object  
  stream object containing channel  
channel  
  name of channel associated with attribute  
attribute  
  name of attribute to create
```

Returns

TRUE on success, FALSE otherwise

Description

This function will delete an attribute associated with a channel in a specific stream object.

Restrictions

Only deletes local attributes; will not delete downstream attributes written previously with `kds_connect_channel` or `kds_resend_attributes`.

C.4.5. `kds_delete_attributes()` — *delete all attributes*

Synopsis

```
int  
kds_delete_attributes(  
  kobject object,  
  kstring channel)
```

Input Arguments

```
object  
  stream object containing channel  
channel  
  name of channel
```

Returns

TRUE on success, FALSE otherwise

Description

This function will delete all attributes from a specified channel within a specific stream object.

Restrictions

Only deletes local attributes; will not delete downstream attributes written previously with `kds_connect_channel` or `kds_resend_attributes`.

C.4.6. `kds_define_attribute()` — *define an attribute*

Synopsis

```
int
kds_define_attribute(
    kstring channel,
    kstring attribute,
    int     datatype,
    size_t  argsize,
    size_t  numargs,
    int     (*get)   (kobject, kstring, kstring, kaddr, kva_list *),
    int     (*set)   (kobject, kstring, kstring, kaddr, kva_list *),
    int     (*match) (kobject, kobject, kstring, kaddr, kaddr, kstring),
    int     (*copy)  (kobject, kobject, kstring, kaddr, kaddr, kstring),
    int     (*query) (kobject, kstring, kstring, size_t *, size_t *, int *, int *),
    int     (*print) (kobject, kstring, kstring, kaddr, kfile *),
    kaddr  clientdata,
    kvalist)
```

Input Arguments

`channel`
name of channel or NULL for object attribute

`attribute`
name of attribute to get

`datatype`
datatype of attribute

`argsize`
size of each attribute argument

`numargs`
number of attribute arguments

`get`
get routine to use instead of default, NULL to use the default.

`set`
set routine to use instead of default, NULL to use the default.

`match`
match routine to use instead of default, NULL to use the default.

`copy`

copy routine to use instead of default, NULL to use the default.
query
query routine to use instead of default, NULL to use the default.
print
print routine to use instead of default, NULL to use the default.
clientdata
client data to pass in to the handler functions
kvalist
open variable argument list to default

Returns

TRUE on success, FALSE otherwise

Description

This function will define an attribute associated with a channel over an entire session.

C.4.7. `kds_resend_attributes()` — *resend all attributes*

Synopsis

```
int  
kds_resend_attributes(  
    kobject object,  
    kstring channel)
```

Input Arguments

object
stream object containing channel
channel
name of channel associated with attributes

Returns

TRUE on success, FALSE otherwise

Description

This function will resend all the attributes from a sending stream for a specified channel.

C.4.8. `kds_get_attribute_names()` — *get attribute names*

Synopsis

```
kstring *
kds_get_attribute_names(
    kobject      object,
    kstring      channel,
    const kstring filter,
    int          *num)
```

Input Arguments

`channel`

name of channel to return attributes for

`filter`

a regular expression to use to as the search key for the attribute name. if it's NULL, all names are returned.

Output Arguments

`num`

number of attribute names

Returns

a string array of attribute names

Description

This function will return the names of all attributes in a specified channel. A filter may be provided to constrain the search.

C.5. Data Casting

Data can be implicitly cast on reading and writing by using the presentation attribute `KDS_PRESENTATION_DATATYPE`.

On a `kds_read_data` call, the data is cast to this datatype from the physical datatype before being returned. On a `kds_write_data` call, the data is cast from this datatype to the actual physical datatype of the channel before being written downstream. In either case, this datatype determines the type of data which should be handled by the user. If not set or set to `KNONE`, no casting will occur.

This attribute only affects the channel in the local process. It is not propagated downstream. This attribute will be copied from a `recv` stream to a `send` stream within a process by a `kds_copy_channel` or

kds_relay_channel call.

The attribute `KDS_COMPLEX_CONVERT` dicates how complex data is handled during casting. If it is converted to a "lower" data type, this attribute specifies how to down-convert the data. For example if the data is actually complex, but the presentation attribute is byte, the complex data would first be converted to the representation defined by this attribute, and then converted to byte.

If the data is being converted from a "lower" data type to a complex data type, this attribute defines how the data should be interpreted as the real or imaginary component of the complex pair. `KPHASE` and `KMAGNITUDE` are invalid values for up converting to complex, and will result in an error.

Reading and writing cast data occur using an internal statically allocated buffer. A new buffer can be set using `kds_set_casting_buffer` function.

C.5.1. kds_set_casting_buffer() — *assign a buffer to use when casting*

Synopsis

```
void
kds_set_casting_buffer(
    kaddr buffer,
    size_t buffer_size)
```

Input Arguments

```
buffer
    buffer to use for casting
buffer_size
    size of buffer to use for casting
```

Description

This function will assign a buffer to use for internally casting data on a `kds_read_data` or `kds_write_data` call. Casting occurs during these calls if the actual datatype of channel differs from the presentation datatype. This buffer will be used for all casting operations done after the buffer is assigned.

If a buffer is not assigned, an statically allocated internal buffer of 1024 bytes will be used. Assignment of a NULL buffer of size 0 will cause streaming data services to revert to using this internal buffer for casting.

An assigned buffer will NOT be freed by streaming data services. The user is responsible for freeing it at the end of the program.

D. Conversion routines

Two kroutines and associated library routines are provided for converting between Polymorphic Data Services and Streaming Data Services. The kroutine *ds2pds* converts from Streaming Data Services to Polymorphic Data Services. *pds2ds* converts from Polymorphic Data Services to Streaming Data Services. Note that streams are supported as a polymorphic format; an incoming stream can be read by any polymorphic data services routine.

E. Sending Example

```
{
    kobject o;

    char *filename = "stream=/tmp/stream.ds"
    int *ldata, *rdata;
    int lnum, rnum;

    /*-- open a new sending stream --*/
    o = kds_open_send(filename);

    /*-- open left channel, set the datatype of the channel, and connect --*/
    kds_open_channel(o, LEFT);
    kds_set_attribute(o, LEFT, KDS_DATATYPE, KINT);
    kds_connect_channel(o, LEFT);

    /*-- open right channel, set the datatype of the channel, and connect --*/
    kds_open_channel(o, RIGHT);
    kds_set_attribute(o, RIGHT, KDS_DATATYPE, KINT);
    kds_connect_channel(o, RIGHT);

    /*-- while we have new left data and right data, send it downstream --*/
    while (ldata = generate_left_data(&lnum) &&
          rdata = generate_right_data(&rnum))
    {
        kds_write_data(o, LEFT, ldata, lnum);
        kds_write_data(o, RIGHT, rdata, rnum);
    }

    /*-- close will close channels for us --*/
    kds_close(o);
}
```

F. Receiving Example

```
{
    kobject i;

    char *filename = "stream=/tmp/stream.ds"
```

```

int    datatype;

int    data[1024];
int    nread;

/*-- open a new receiving stream --*/
i = kds_open_rcv(filename);

/*-- select the next available channel --*/
while (channel = kds_select_channel(i, &num))
{
    kds_get_attribute(i, channel, KDS_DATATYPE, &datatype);

    /*-- only process integer data from the left channel --*/
    if ((datatype == KINT) && (kstrcmp(channel, LEFT) == 0))
    {
        /*-- read no more than 1024 bytes --*/
        nread = kds_read_data(i, channel, (kaddr) data, 1024);

        for (i = 0; i < nread; i++)
            kprintf("%d\\", data[i]);
    }

    /*-- other data should just be discarded --*/
    else
        kds_copy_data(i, channel, NULL);
}

kds_close(i);
}

```

G. Sending and Receiving Example

```

{
    int pending, datatype, nread;
    kstring in_file = "stream=/temp/in.ds";
    kstring out_file = "stream=/temp/out.ds";
    kstring ch_in, *ch_out = "out";
    kobject src, dest;
    kaddr *data;

    /*-- open new sending and receiving streams --*/
    src = kds_open_rcv(in_file);
    dest = kds_open_send(out_file);

    /*-- select the next available channel --*/
    while ((ch_in = kds_select_channel(src, &pending)) != NULL)
    {
        if (pending == -1)
        {
            /*-- Copy the datatype and open and connect downstream channel --*/
            kds_get_attribute(src, ch_in, KDS_DATATYPE, &datatype);
            kds_relay_channel(src, ch_in, dest);
        }
        else
        {

```

```
data = kmalloc(pending * kdata_size(datatype);
nread = kds_read_data(src, ch_in, data, pending);

/*-- Processing data goes here --*/

kds_write_data(dest, ch_out, data, nread);
kfree(data);
}
}

kds_close(src);
kds_close(dest);
}
```

This page left intentionally blank

Table of Contents

A. Introduction	7-1
B. Streaming Data Model	7-2
C. Stream Functions	7-4
C.1. Sending Stream Functions	7-4
C.1.1. <code>kds_open_send()</code> — <i>open a sending stream</i>	7-5
C.1.2. <code>kds_open_channel()</code> — <i>open a sending channel</i>	7-6
C.1.3. <code>kds_connect_channel()</code> — <i>connect an sending channel</i>	7-6
C.1.4. <code>kds_write_data()</code> — <i>write data down a sending channel</i>	7-7
C.1.5. <code>kds_close_channel()</code> — <i>close a sending channel</i>	7-8
C.1.6. <code>kds_close()</code> — <i>close a stream</i>	7-8
C.2. Receiving Stream Functions	7-9
C.2.1. <code>kds_open_recv()</code> — <i>open an receiving stream</i>	7-9
C.2.2. <code>kds_select_channel()</code> — <i>select next receiving channel with available data</i>	7-10
C.2.3. <code>kds_read_data()</code> — <i>read data from a receiving channel</i>	7-10
C.3. Sending to Receiving Functions	7-11
C.3.1. <code>kds_query_channel()</code> — <i>query a channel</i>	7-11
C.3.2. <code>kds_relay_channel()</code> — <i>relay a channel to a sending stream</i>	7-12
C.3.3. <code>kds_copy_channel()</code> — <i>copy a channel and associated data</i>	7-12
C.4. Attribute Functions	7-13
C.4.1. <code>kds_set_attribute()</code> — <i>set an attribute</i>	7-13
C.4.2. <code>kds_get_attribute()</code> — <i>get an attribute</i>	7-14
C.4.3. <code>kds_create_attribute()</code> — <i>create an attribute</i>	7-15
C.4.4. <code>kds_delete_attribute()</code> — <i>delete an attribute</i>	7-15
C.4.5. <code>kds_delete_attributes()</code> — <i>delete all attributes</i>	7-16
C.4.6. <code>kds_define_attribute()</code> — <i>define an attribute</i>	7-17
C.4.7. <code>kds_resend_attributes()</code> — <i>resend all attributes</i>	7-18
C.4.8. <code>kds_get_attribute_names()</code> — <i>get attribute names</i>	7-19
C.5. Data Casting	7-19
C.5.1. <code>kds_set_casting_buffer()</code> — <i>assign a buffer to use when casting</i>	7-20
D. Conversion routines	7-21
E. Sending Example	7-21
F. Receiving Example	7-21
G. Sending and Receiving Example	7-22

This page left intentionally blank