Created:January 17, 1996
Modified:March 6, 2000

Gemini Data Handling System Report

ICD 3 — Bulk Data Transfer

Norman Hill, Séverin Gaudet

dhs_pdr_icd3/26

This document defines the interfaces in the Gemini System when transferring bulk data between principal systems and the DHS.

1.0 Introduction

This document defines the interfaces in the Gemini System when transferring bulk data between principal systems and the DHS. The ICD 3 interface is built on the ICD 1c interface described in [6], which should be read before this document.

1.1 Purpose

The Gemini *Software Design Description* [1] contains a design for the major Gemini systems. An overview describing all the interfaces between the Gemini systems is given in document [3] (See the list of references on page 2 for more background material).

This document (ICD 3 — Bulk Data Transfer) specifies those interfaces in the Gemini system which involve the transfer of bulk data from one principal system to another. The interfaces involved are:

- The transfer of *pixel and header data* from the Instrument Control System (ICS) or any other Gemini system to the Data Handling System (DHS).
- The transfer of *header data* from the Observatory Control System (OCS) to the DHS.
- The transfer of *pixel and header data* from the DHS to other Gemini systems.
- The transfer of *pixel and header data* from the DHS to the Quick Look Displays.

All the above data flows boil down to the transfer of data intended for storage to disk and/or Quick Look Display and the transfer of data from the DHS to other Gemini systems. These flows are handled by standard functions and data structures, described later.

1.2 Scope

1.2.1 What is covered

The document covers *only* bulk data transferred internally between the principal systems, as listed in “Purpose” on page 1.

1.2.2 What is not covered

The document does not cover the transfer of data to external or private entities, such as:

1. The transfer of data from an ICS to a real-time video display.
2. The storage of data to a private disk by an ICS.
3. The transfer of raw science data from the DHS to the external data reduction system.
4. The transfer of reduced data from the external data reduction system to the DHS.
5. The transfer of status information and data from the DHS to the permanent store and transport media.
6. The transfer of data between sub-systems.

These flows are or will be covered by other Gemini documents.

This ICD does not constrain how bulk data is handled internally by the various work packages; it just describes how that data is transferred across work package boundaries. As an example, a Detector array Controller may need to transfer data internally as C structures using a transputer link protocol. The data only has to meet the requirements of this document when it is shipped out from the Detector array Controller to the ICS or DHS.

1.3 References

The following documents should also be consulted:

[1] *Gemini Software Design Description*, SPE-C-G0037, Gemini 8m Telescopes Project

[2] *Gemini Software Programming Standards,* SPE-C-G0009/02, Peregrine M.McGehee, Steve Wampler

[3] *Gemini System Interfaces*, GSCG.grp.005, Gemini 8m Telescopes Project

[4] *ICD 1a — The System Command Interface*, GSCG.KKG.009/009, Kim Gillies, Gemini 8m Telescopes Project

[5] *ICD 1b - The Baseline Attribute/Value Interface*, GSCG.grp.024, Kim Gillies, Steve Wampler, Bret Goodrich, Gemini 8m Telescopes Project

[6] I*CD 1c — Baseline DHS interface*, icd1c/05, Norman Hill, Séverin Gaudet and Dayle Kotturi, HIA

[7] *ICD 3.2 — The DHS Command Interface*, dhs_icd3.2/05, Norman Hill, Dayle Kotturi and Séverin Gaudet, HIA

[8] *ICD 2 — Systems Status and Alarm Interface*, Version GSCG.grp.013-ICD2/005, Kim Gillies, Gemini 8m Telescopes Project

[9] *ICD 4 —Logging Information*, dhs_pdr_icd4/11, Norman Hill and Séverin Gaudet, HIA

[10] *ICD 8 — Non-Conforming ICS Interfaces*, GSCG.grp.012, Gemini 8m Telescopes Project

[11] G*lossary for the Gemini Software*, GSCG.grp.016, Gemini 8m Telescopes Project

[12] *Report on the OCS/DHS Interface Meeting*, gscg.sbw.082, Gemini 8m Telescopes Project

[13] *Guide to Writing DRAMA Tasks,* AAO/DRAMA_GUIDE_3, Tony Farrell, Anglo-Australian Observatory

[14] *Self-defining Data System (SDS)*, AAO/SDS_07, Jeremy Bailey, Anglo-Australian Observatory, Version 1.2, 9 September, 1992

[15] *Interprocess Message Passing (IMP) System*, AAO/IMP_MANUAL_8, DRAMA Software Report 8, Version 1.3, 13 June 1997, Keith Shortridge, Anglo-Australian Observatory

[16] *Definition of the Flexible Image Transport System (FITS)*, Proposed Standard, NOST 100-1.0, March 8, 1993, NASA Science Office of Standards and Technology

[17] *NDF — Routines for Accessing the Extensible N-Dimensional Data Format*, SUN/33, Rodney Warren-Smith, Starlink Project, Rutherford Appleton Laboratory, UK

1.4 Abbreviations and Acronyms

A&G	Acquisition and Guidance subsystem of the TCS.
AAO	Anglo-Australian Observatory.
AO	Adaptive Optics sub-system of the TCS.
ARG	Argument structure processing routines. Part of SDS.
ASCII	American Standard Code for Information Interchange.
BNF	Backus Naur Form.
CCD	Charge Couple Device.
DHS	Data Handling System.
DRAMA	Distributed Real-time AAO Monitor for Astronomy.
DS	Data Server.
FITS	Flexible Image Transport System.
GCS	Gemini Control System.
GSCG	Gemini Software and Controls Group.
ICD	Interface Control Document.
ICS	Instrument Control System.
IMP	Interprocess Message Passing System.
LAN	Local Area Network.
N/A	Not Applicable.
OCS	Observatory Control System.
PDF	Parameter Description File.
SDD	Software Design Description.
SDS	Self-defining Data System.
TCP/IP	Transmission Control Protocol/Internet Protocol.
TCS	Telescope Control System.
VME	A real-time system obeying the ANSI/IEEE 1014-1987 Versatile Backplane Bus standard.
WFS	Wave Front Sensor.

1.5 Glossary

Attribute/keyword dictionary — A database of attributes, and the properties of those attributes. The DHS Attribute/keyword dictionary will contain all header attributes transmitted with the bulk data interface.

Attribute/value pairs — Header data consists of attributes (i.e. airTemperature, ra, observer) and values (i.e. 22.5, 10:9:8.7, “Joe Astronomer”). Valid attributes are defined in the DHS attribute/keyword dictionary.

Backus Naur Form — A method of defining the relationship between data objects using a series of short-hand symbols to represent combination, choices and repetition. The symbols are:

=	“consists of”
+	“and” (sequence)
[|]	“or” (selection)
{ }	“repeating” (iteration)
()	“optional components”
@	“identifier, key attribute”
-ref	“reference to (key of) another data object”
;	“end of sentence”

Bulk data — FITS data, instrument datasets, or raw binary data to be displayed or stored to magnetic disk.

Client system — A Gemini principal system using this interface to send commands or data to the DHS and to receive any resulting responses.

Data Chunk — A subset of the data comprising a complete dataset. This may include a subset of the header data, and/or one or more sub-rasters of the data arrays.

Data LAN — Local Area Network in the Gemini system devoted to the transfer of bulk data.

Data Server — A process in the DHS which receives data from or transmits data to a remote client.

Dataset — A dataset is the data resulting from an OBSERVE command. A dataset consists of header data describing the dataset and one or more frames.

Dataset Name — A name that identifies a dataset.

DHS library — A library supplied by the DHS work package containing the public programmatic interface to the DHS system.

Display Tool — A process for displaying data in the DHS. Display tools are created and controlled by observers or observatory staff, or by Gemini applications. See also Quick Look Display.

Exposure — An array of data, collected between detector resets, representing the result of one exposure of the detector to light. This may be as a result of a single detector read in destructive read mode or a combination of detector reads in non-destructive read mode.

FITS data buffer — A memory buffer containing data exactly as it appears in a FITS data file.

FITS format file — A file obeying the Flexible Image Transport system standard, as defined in [16]. In this document the term “FITS format file” will refer to a file written to disk.

FITS header — A FITS header as written to a FITS format file. This consists of a series of 80-byte ASCII card images. Each record contains an 8 character keyword identifying the item, the value of the item encoded as ASCII characters, and an optional comment describing the item (see [16]).

Frame — Header data and a single array of pixel data. The frame can represent a single exposure or combined exposures. It can also be bad pixel masks, quality data, variance data, etc.

Frame region — A rectangular subset of a frame's data array (synonymous with sub-raster).

Header data — A collection of attribute/value pairs which describe a dataset or a frame within a dataset. The attributes must be described in the DHS attribute/keyword dictionary.

Hierarchical data structure — A data structure built like a Unix directory structure, where related items of data can be grouped together into structures, and the structures themselves can be grouped into more complicated structures. The whole data structure resembles a tree, with the data units themselves appearing as leaves on the tree. SDS allows the definition of hierarchical data structures.

numAxes — The number of axes a particular frame has. A 1-D spectrum would have 1 axis, a 2-D image 2 axes, and a 3-D data cube 3 axes. SDS can handle data with up to 7 axes.

Normal observation — This phrase refers to observations taken under the control of the OCS where the instrument supplies the pixel data and most of the header data, with the remainder of the header data being supplied by OCS. Other types of observations are not excluded, but this is the normal operating mode of the Gemini telescopes.

Permanent data — Bulk data sent to the DHS via the ICD 3 interface which will eventually be written to the Gemini permanent store.

Pixel data — Raw or processed pixel data from an instrument's detector.

Principal system — At the highest level in the GCS software decomposition, the software system is divided into four systems called principal systems. The systems are called: the Data Handling System, the Observatory Control System, the Telescope Control System, and the Instrument Control System. There may be one or more concurrently executing Instrument Control Systems.

Primitive data type — The type of data representing a "leaf" in a hierarchical data structure. A primitive data type will be one of the types recognized by programming languages (such as "integer", "floating point", "integer array" etc.

Processed data — Bulk data generated by processing raw data with a data processing recipe. This can be science data or calibration data.

Quick Look Display — An instance of a display tool used to display data soon after it has been collected by an instrument. Quick look displays are started by observers or observatory staff and are used to display the data being received by the DHS Data server.

Quick look stream — The connection between Quick Look Display and bulk data client systems is made using Quick Look streams. A Quick Look stream is a character string which a client system associates with data. Quick Look Displays can then register interest in Quick Look streams, and the DHS can then route data to appropriate Quick Look Displays as it is received.

Server system — System responsible for responding to commands sent over this interface. The server system will be a part of the DHS system.

Temporary data — Bulk data sent to the DHS Data Server which will not be written to the Gemini permanent store, and which is deleted when the DHS system is shut down or reset. Temporary data is otherwise identical to permanent data.

Transient data — Bulk data sent to the DHS Data Server which is not saved to any storage device. Transient data would only be forwarded to any interested Quick Look Displays.

Unique Name — A unique name assigned by the DHS or OCS which is used to identify data being sent to or requested from the DHS. A unique name can be used to uniquely identify the data for as long as the data exists. (Section 8.0 on page 35.)

Unix — A standard host-level operating system. The Gemini baseline for “Unix” is “Solaris” marketed by Sun Microsystems.

VxWorks — A real-time operating system for VME hardware marketed by Wind River Systems.

1.6 Stylistic Conventions

1.6.1 References to other documents.

References to other documents are given using a number in square brackets, for example [1]. The exact references are given in Section 1.3 on page 2.

1.6.2 References to the Gemini systems

The term “ICS” is used in statements which refer equally well to any Instrument Control System. Other Gemini systems are referenced by their acronyms as they appear in Section 1.4 on page 3.

1.6.3 Functions and parameters

Function and parameter names are given in mixed case as specified in [2]. Data types are written in upper case. Functions are described by showing their ANSI C prototype definitions.

1.6.4 Representation of data structures

In the data structure listings (page 28 onwards), the structure names and data types are given in upper case. Generic types are given in lower case. Names of data items are given in mixed case.The contents of character strings and descriptions of the items are given in mixed case, the descriptions being enclosed in square brackets. Data types are enclosed in angle brackets. I have used the suffix “()” to show that a data item is an array of specifiable size and “(…)” to show a series of arrays which are all the same size. The syntax (i..j) is used to specify elements i through j of an array. Data structure listings are indented to show their hierarchy.

1.6.5 Numeric indexes

Since C is the standard language for the Gemini Project, and since all C indexes are zero based, all numeric indexes in this document are also zero based.

1.6.6 Typographic conventions

Program text is shown in the `Courier` type face. References to symbols in the DHS library are shown in **bold** type face.

2.0 Overview

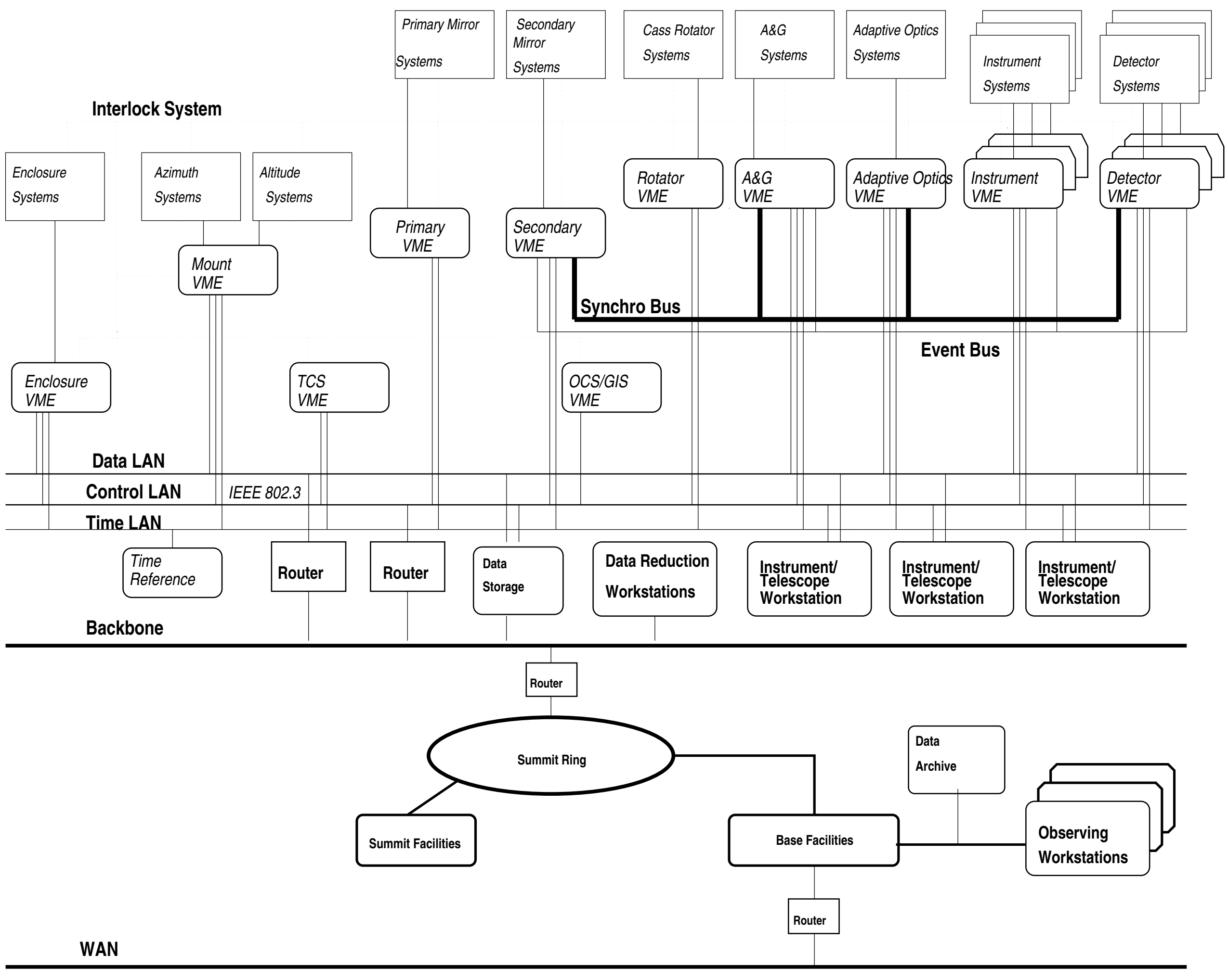
The Gemini Control System (GCS) is made up of principal software systems: the Telescope Control System (TCS), the Data Handling System (DHS), the Observatory Control System (OCS), and one or more Instrument Control Systems (ICS). The purpose of these systems, and the way they interact with each other, is described in the SDD [1].

ICD 3 describes a mechanism by which any system in the GSC can send bulk data to the DHS. Document [3] gives a summary of the data flows between the principal systems and indicates which flows are covered by this ICD.

2.1 System Hardware Architecture

The overall layout of the computer hardware and communication equipment is shown in Figure 1 on page 7 which is taken from [3] and modified to reflect recent design decisions.

FIGURE 1. The System Hardware Architecture



2.1.1 The Instrument Control Systems (ICS)

A “conforming” ICS is VME-based running VxWorks and EPICS.

A “non-conforming” ICS is based on some other architecture without access to EPICS or VxWorks.

This document assumes the ICS architecture is conforming. See [10] for a description of the interface to a non-conforming ICS.

2.1.2 The Data Handling System (DHS)

The DHS is assumed to be based on a UNIX work station (baselined as a Sun Sparcstation running Solaris 2.X). No other DHS architecture is currently supported.

2.1.3 The Observatory Control System (OCS)

The OCS is assumed to be based on a UNIX work station (baselined as a Sun Sparcstation running Solaris 2.X). No other OCS architecture is currently supported.

2.2 Communication Architecture

Generally for bulk data transfer, the DHS will communicate over the Gemini Data LAN using a yet-to-be-determined LAN technology, and DHS commands will be sent over the control LAN, but both bulk data and commands can be carried by any TCP/IP LAN. The selection of the LAN used for any connection will be done when the client executes the **dhsConnect** function described in [6].

2.3 Context Diagram

Figure 2 on page 9 shows a context diagram as seen by the bulk data interface. The Gemini systems appear to the interface as entities in its outside world. The diagram may be compared to the "Gemini Control System" data flow diagrams contained in [1].

The relationship between this context diagram and the data flow diagrams for the "Gemini Control System" is as follows:

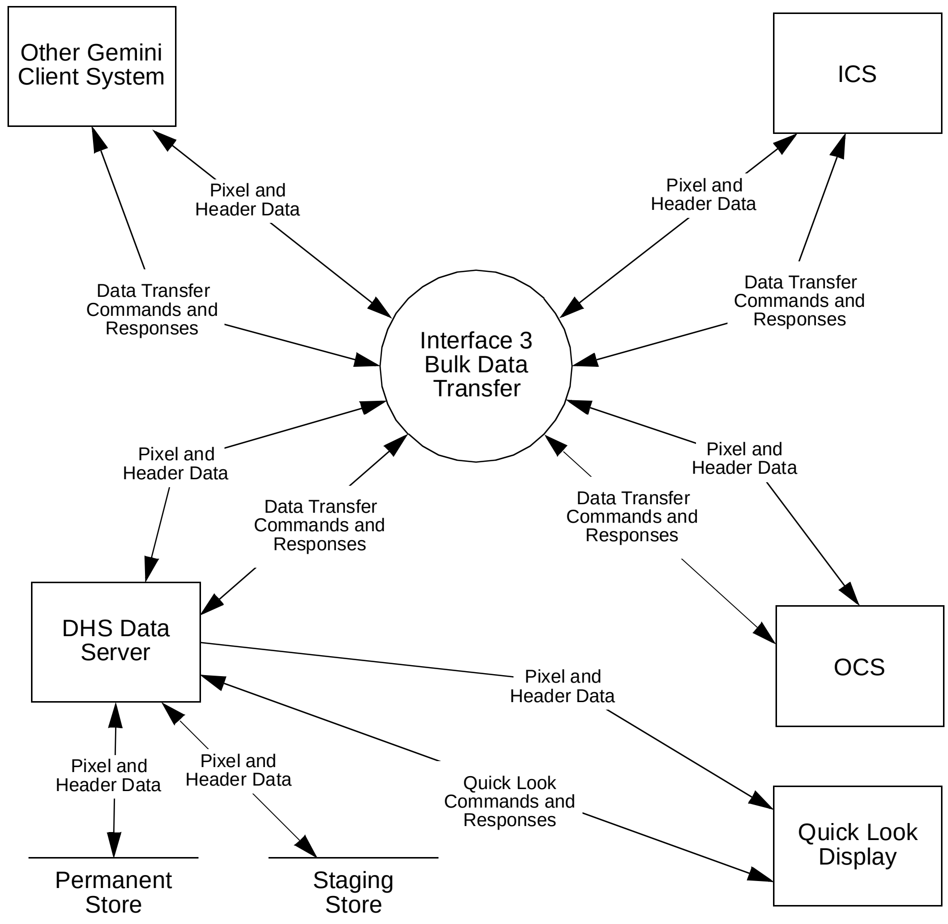
- "Internal Data Processor" and "Instrument Sequencer" are part of the Instrument Control System and may be found in data flow diagram 2.
- "Configurable Control System" is part of the Observatory Control System, and may be found in data flow diagram 1.
- "Quick Look Display" represents the processes "DHS/DC Quick Look Server", "DHS/ICS Quick Look Server", "DHS/OCS Quick Look Server" and "DHS/DHS Quick Look Server", which are contained in data flow diagrams 2.3, 2, 1.3 and 4.3 respectively. All these processes can be implemented using the same Quick Look Display design, which takes "Raw Data", 'Reduced Data" or a "Raw Frame" and displays it together with textual information. The implementation of this server is part of the DHS work package.

3.0 Behaviour

The DHS work package has provided a C library which can be linked with UNIX or VxWorks programs to allow Gemini Principal systems to communicate with the DHS and its components. This library will be referred to as the DHS library in the rest of this document. The DHS library allows clients to establish connections to data servers, create and decode dataset structures (as described in Section 7.0 on page 28), and to send and receive dataset structures from a DHS data server.

There are two fundamentally different modes in which the ICD 3 interface can be used. The primary mode of use involves setting application supplied callback functions which the DHS library

FIGURE 2. Context Diagram



will call when various events occur. The callback functions are responsible for correctly dealing with the events. In this mode, the application would start a DHS library event loop which then runs continuously, executing callback functions as required. An option is provided to allow the application to run the event loop in a thread, freeing the application to perform other tasks.

In the second mode of use, client systems make ICD 3 data transfer requests and then either poll the status of the requests or wait for commands to complete and then deal with the responses to the requests. Using this mode, the DHS library will run the event loop whenever it is required.

The interface to the DHS library is the same for both modes with the callback mode being used when the client runs the event loop. The two modes are not mutually exclusive, so it is possible for an application to wait for requests and query request status even if callbacks functions are being used. (I'm not sure why anyone would want to mix modes, but there is no reason to prevent it.)

The details of the public interface to this library are described in Section 5.0 on page 12. All functions of the DHS library are thread safe.

3.1 Events and Responses

The events which may occur on the bulk data transfer interface, and the responses to those events, are shown in Table 1 on page 10.

TABLE 1. DHS Data Sever Events and Responses

Event	Response
ICD 3 client connects to an ICD 3 server.	The connection is recorded within the server and a **DHS_CONNECT** is returned.
ICD 3 client disconnects from an ICD 3 server.	The connection is closed by the server. The **DHS_CONNECT** becomes invalid.
ICD 3 client requests a unique name from the ICD 3 server.	A unique name of appropriate type is generated and returned to the requesting system.
ICD 3 client system sends data to the ICD 3 server system.	The library function returns immediately giving a **DHS_TAG** to the application. The server receives the data and performs any necessary actions. The client call-back routine is executed when all actions are complete.
ICD 3 client waits for data to arrive at the ICD 3 server. (Optional.)	Control is returned to the client when the specified data has been received and processed by the server.
ICD 3 client queries the current status of data sent to a ICD 3 server. (Optional.)	The current status of the specified data is returned to the client.
ICD 3 client requests data from an ICD 3 server.	The requested data is returned to the client. A callback function is executed when the data is received.
ICD 3 client issues an abort command.	All data for the specified dataset name is discarded by the server.
ICD 3 client issues a reset command.	The data for the specified dataset name is reset to the null value by the server.
ICD 3 client loses server connection (server crashes).	The DHS library error callback function will be executed, and the client application can attempt to re-establish its connection to the server.
ICD 3 server loses connection with client (client crashes).	The server will close the connection to the client and wait for the client to re-connect. An error will be issued if any data was lost due to the dropped connection. After the connection is re-established, the client may continue sending data for a unique name that was in use when the connection was lost.

The following sections describe the normal behaviour of the system.

3.1.1 When a new instrument is commissioned (i.e. when it is installed on the system for the first time)

The list of header attributes generated by an instrument is entered into the DHS attribute/keyword dictionary from the instrument PDF. The properties of the header attributes entered into the attribute/keyword dictionary will include: attribute name, data type[1], valid range, is the attribute mandatory, specification of how the data will be stored in the FITS file (e.g. attributes that are

1. A special type will be created to refer to a reference file name. This will allow the DHS to check that all reference files are available.

arrays could be stored as binary FITS tables, ASCII fits tables, images, or indexed keywords), the instrument modes where the attribute applies and the null value.

3.1.2 At system start-up

The OCS, ICSs and DHS are all started. Systems which will be sending bulk data to or receiving bulk data from the DHS will probably initialize a connection to the DHS, however this could be delayed until a later time.

3.1.3 When a new observer registers to use an instrument

The new observer starts Quick Look Displays as required. The observer can use more than one display if required. The Quick Look Displays will be identical but the observer can configure them differently and use them to display data from different Quick Look streams.

3.1.4 When a dataset is created (i.e. An observe sequence is executed)

The following sequence of events occurs when a dataset is created:

1. Either DHS or OCS assign a unique name for the data. The ICD 3 interface includes functionality to return unique names to any requesting system. All client systems sending data for the dataset must use this unique name to send data to the DHS. A client system can send more than one dataset for each unique name by incrementing the group-ids as described in Section 8.0 on page 35.
2. All systems with data to contribute to the dataset should send the data. The order in which the data is sent isn't critical, however header information should be sent as early as possible to allow the DHS to use the information on data displays. Subsets of both pixel and header data can be sent separately if necessary or desired. If more than one frame is to be sent to the DHS, the frames may be sent together or separately and in any order.
3. If more than one system will be contributing data for the dataset then the list of contributors must be set with the **dhsBdCtl** function.

3.1.5 When a Gemini system requests data from a file (i.e. The system uses the dsGet function)

The DHS returns the requested data to the requesting system. Any Gemini system which needs access to the data stored can request the data at any time, however if possible, data requests should be performed during system initialization to avoid bandwidth contention with data from observations.

3.1.6 At system shutdown

All systems disconnect from the DHS. If data has been received for a dataset but is not yet complete, an error is issued.

3.1.7 Recovery from crashes and power failures

The interface is designed to be fault-tolerant. If either the client or server dies an attempt should be made to re-establish a connection. For example:

- If the client dies and Data Server continues running. When the client restarts it should attempt to re-connect to the Data Server.
- If the Data Server dies the interface will detect that the server is no longer available and execute the client systems error callback function. The client application can then attempt to re-establish its connection to the server. Add datasets that are fully complete are guaranteed to be saved to permanent store. All data chunks the Data Server has acknowledged as being received will be saved to permanent store in a raw form when the Data Server recovers. Any

incomplete datasets must be re-sent by the client, or manually assembled from the incomplete chunks stored by the Data Server.

3.1.8 A Gemini system needs to store data or display data with Quick Look

At any time, any Gemini system can get a unique name from the DHS and send data for permanent storage, temporary storage or Quick Look Display. This facility could be used by the A&G system to store or display data, and may be used by instruments during an observe sequence to display data not directly related to the observation (for example the output of on-board wave front sensors could be sent to a Quick Look Display).

4.0 Implementation

The ICD 3 interface is based on the ICD 1c Baseline DHS Interface described in [6], which is in turn based on the DRAMA IMP messages passing system [15], and DRAMA SDS data structures[14]. The data structures use to contain the frame data are also be DRAMA SDS data structures.

5.0 The Programmatic Interface

All public symbols in the DHS library will be prefixed by the string “dhs” or “DHS”, and public symbols that apply only to bulk data transfer will be prefixed by dhsBd or DHS_BD. Typical uses of the ICD 3 interface are described in Section 5.1 on page 12 and Section 5.2 on page 13 and example code is given in Section 10.0 on page 37. Although the interface is described in terms of “client” use of the library and “server” use of the library, the interface is basically bidirectional, and it is possible for “servers” to send data to “clients” if the appropriate callback routines have been set up by the client, although the only uses of this feature we foresee are internal to the DHS. The only clear distinction between client and server is that the client is the one that uses the **dhsConnect** function call to establish the initial connection.

5.1 Client use of the interface

Typically, a client system using the ICD 3 interface to send data to the DHS data server would:

1. Initialize the library with **dhsInit**.
2. Set any callback functions with the **dhsCallbackSet** function. The error callback must be set at a minimum, and most clients will set either the get or the put callback functions.
3. Make one or more connections to DHS servers with the **dhsConnect** function. These connections should preferably be over the Gemini data LAN.
4. Probably start the event loop in a thread with the **dhsEventLoop** function. If no event loop is started, the DHS library will run the event loop when it is has control and is waiting for events.
5. Use the data formatting commands described in Section 6.2 on page 19 to create a data structure.
6. Use the **dhsBdPut** function to send the data structure to the DHS data server.
7. Use the **dhsWait** and **dhsStatus** functions to monitor the progress of data, or continue processing and allow callbacks to be called depending on the use of the library.
8. In a callback routine, or after polling to ensure the transfer is complete, the client would clean up after the transmission and use the **dhsTagFree** function to destroy the **DHS_TAG**.

9. Close connections with the **dhsDisconnect** function.
10. Stop the event loop with the **dhsEventLoopEnd** function.
11. Clean up the library with the **dhsExit** function.

Generally steps 1 through 4 are initialization, and steps 9 through 11 are cleanup, although the library may be initialized and exited repeatedly, callback functions may be set or changed at any time, connections may be opened and closed as required, and the event loop may be started and stopped at any time. If steps 9 and 10 are omitted, they will be performed automatically by the **dhsExit** function, but I think it is more symmetrical to explicitly end things that are explicitly started. Note than much of the initialization and cleanup is shared with that required by the ICD 1c DHS command interface.

A user data pointer may be provided as a parameter to the **dhsBdPut** and **dhsBdGet** functions. If provided by the application, the user data pointer will be associated with the command tag. User data is not passed to the server or interpreted by the DHS library in any way, but is retrievable with the **dhsUserDataGet** function and will be passed as an argument to the callback functions. The purpose of the user data pointer is to allow an application to supply a context which is easily available when handling responses. The user data will probably consist of a pointer to a application supplied C structure, C++ object, or function (C or C++), but the use of this pointer is entirely up to the application programmer and may be any construct that may be referred to by a void pointer. The user data pointer provides all of the opportunities for interesting bugs associated with this style of programming (passing pointers to data that may or may not be invalid when the callback is executed, etc.) and should be used with caution.

5.2 Server use of the interface

Typically, a server using the library would:

1. Initialize the library with the **dhsInit** function.
2. Set any callback functions with the **dhsCallbackSet** function. The error, server get and server put callback functions should all be set and a command callback should be set up to handle control commands (see ICD 1c [6]).
3. Probably start the event loop with the **dhsEventLoop** function, but not in a separate thread. Most servers will not be required to perform any tasks between servicing requests so it would not be necessary to have a separate thread for the event loop.
4. The server callback routines will be executed as client systems make data requests.
5. When some exit condition is met the **dhsEventLoopEnd** function should be used to stop the event loop.
6. Clean up the library with the **dhsExit** function.

Steps 1 through 3 are initialization, and steps 5 and 6 are cleanup.

5.3 Interface summary

The data types used by the ICD 3 interface are summarized in Table 2 on page 14 and the ICD 3 interface functions are summarized in Table 3 on page 14. The reference column in the tables indicates the section of this document where a detailed description of the item will be found. References to [6] are items which are described in detail in [6] and not in this document.

TABLE 2. ICD 3 data type summary

Data Type	Description	Reference
DHS_BD_CTL	Indicates type of control function	6.1.1
DHS_BD_DATASET	Value identifying a dataset structure	6.1.2
DHS_BD_FRAME	Value identifying a frame	6.1.3
DHS_BD_ATRIB_ID	A value identifying an attribute in a frame or dataset	
DHS_BD_GET_TYPE	Indicates the format for retrieved data	6.1.4
DHS_BD_LIFETIME	Lifetime of the data associated with a dataset name	6.1.5
DHS_BD_PUT_TYPE	Type of data being sent	6.1.6
DHS_CB_FN_PTR	Pointer to a callback function	[6]
DHS_CB_TYPE	Type of callback function	6.1.7, [6]
DHS_CONNECT	Identifies a client or server connection	[6]
DHS_CON_STATE	The current state of a connection	[6]
DHS_DATA_TYPE	Type of a data item	[6]
DHS_DEBUG_LEVEL	Debugging level of the DHS library	[6]
DHS_EL_TYPE	DHS library event loop type	[6]
DHS_ERR_LEVEL	Error level of an error status	[6]
DHS_STATUS	Status value returned by functions in the DHS library	6.1.8, [6]
DHS_TAG	Identifies a specific command	[6]

TABLE 3. ICD 3 interface function summary

Function	Description	Reference
General functions		
dhsInit	Initialize the DHS library	[6]
dhsExit	Clean up the DHS library	[6]
dhsMessage	Get the DHS library error numbers and level, and a pointer to the message string	[6]
dhsMessageClear	Clear a DHS message	[6]
dhsEventLoop	Start the DHS event loop	[6]
dhsEventLoopEnd	Stop the event loop	[6]
dhsCallbackSet	Set a DHS library callback function	[6]
dhsDisconnect	Used by a client to disconnect from a server, or by a server to refuse a connection from a client	[6]
dhsDebugLevel	Set the debugging level of the DHS library	[6]
dhsElBroadcast	Cause the event loop to execute one no-op iteration	[6]
dhsIsConnected	Test to see if a connection is open and useable.	[6]
Client functions		

TABLE 3. ICD 3 interface function summary (Continued)

Function	Description	Reference
dhsConnect	Connect to a server	[6]
dhsConUserDataGet	Get a connection's user data pointer	[6]
dhsConUserDataSet	Set a connection's user data pointer	[6]
dhsConnectInfo	Returns information about the other end of a connection	[6]
dhsWait	Wait for a list of commands to complete	[6]
dhsStatus	Query the current status of a command	[6]
dhsTagFree	Free a command tag	[6]
dhsUserDataGet	Get the user data pointer associated with a command	[6]
dhsUserDataSet	Set the user data pointer associated with a command	[6]
dhsTagDone	Check to see if a tag is in one of the end states	[6]
Bulk data formatting functions		
dhsBdDsNew	Create a new DHS_BD_DATASET structure	6.2.1
dhsBdDsFree	Free the memory allocated to a dataset structure	6.2.2
dhsBdFrameNew	Add a new frame to a dataset structure	6.2.3
dhsBdFrameInfo	Get information about a frame	6.2.4
dhsBdFrameFind	Find a frame by name	6.2.5
dhsBdFrameIndex	Find a frame based on its position in the dataset structure	6.2.6
dhsBdAttribAdd	Add an attribute to a frame or dataset structure	6.2.7
dhsBdAttribDelete	Delete an attribute and its value from a dataset or frame	6.2.8
dhsBdAttribInfo	Get information about an attribute	6.2.9
dhsBdAttribFind	Find an attribute by name	6.2.10
dhsBdAttribIndex	Find an attribute based on its position in a frame or dataset	6.2.11
dhsBdDsAccess	Access an dataset in a memory buffer	6.2.12
dhsBdDsCopy	Copy a dataset	6.2.13
dhsBdDsExport	Export a dataset to a memory buffer	6.2.14
dhsBdDsPrint	Print a dataset to standard out	6.2.15
dhsBdDsSize	Return the size of a dataset	6.2.16
Client bulk data transfer functions		
dhsBdCtl	Send a control command to the Data Server	6.3.1
dhsBdName	Retrieve a new unique name from the Data Server	6.3.2
dhsBdGet	Get the data for a dataset name	6.3.3
dhsBdPut	Send a dataset or chunk of a dataset to the Data Server	6.3.4
dhsBdDelete	Delete a dataset from a Data Server	6.3.5
Server bulk data transfer functions		
dhsBdResponse	Send a response containing bulk data to a client	6.4.1

TABLE 3. ICD 3 interface function summary (Continued)

Function	Description	Reference
User supplied functions		
command callback	Function called on server when receiving ICD 1c commands	[6]
error callback	User supplied error callback function	[6]
connect callback	Function called when a connection state changes.	[6]
get callback	Function called on the client when requested data is available	6.5.1
put callback	Function called on the client when data storage is complete	6.5.2
server get callback	Function called on the server system when data is requested	6.5.3
server put callback	Function called on the server system when data is sent	6.5.4

6.0 Detailed Interface Description

The following sections detail the public ICD 3 interface. The functions are separated into sections based on the situations where a function would normally be used. It is assumed that the header file "dhs.h" has been included before any of the DHS library functions are used. All DHS library functions have an output parameter of type **DHS_STATUS** as the last parameter, which returns the completion status of the function. After successful completion a function will return the value **DHS_S_SUCCESS**. If any function is called with the status value not equal to **DHS_S_SUCCESS**, the function will return immediately without modifying the status value or performing any actions. The return status values are defined in dhs.h and listed in [6]. The return status values that apply to the ICD 3 interface are listed in Table 4 on page 18.

6.1 Data types

The following data types are defined in file **dhs.h**, and used by the ICD 3 interface:

6.1.1 DHS_BD_CTL Indicates type of control function

This is an enumerated data type used to indicate which control function should be performed when function **dhsBdCtl** is called. Possible values for this data type are:

DHS_BD_CTL_ABORT — Abort a dataset transfer.

DHS_BD_CTL_RESET — Reset all of a partially received dataset to an empty state.

DHS_BD_CTL_LIFETIME — Change the lifetime assigned to a dataset name.

DHS_BD_CTL_CONTRIB — Change the list of contributors to a dataset name.

DHS_BD_CTL_QLSTREAM — Change the list of Quick Look streams associated with a dataset name.

DHS_BD_CTL_GETNAME — Get a new unique name from the data server. This control is used by the **dhsBdName** function, and may not need to be used directly.

6.1.2 DHS_BD_DATASET Value identifying a dataset structure

This is a value that is used to refer to the data for a dataset. The structure of a **DHS_BD_DATASET** is described in Section 7.0 on page 28. This structure is manipulated with the bulk data formatting functions described in Section 6.2 on page 19.

6.1.3 DHS_BD_FRAME Value identifying a frame

This is a value that is used to refer to a frame in a dataset structure. The structure of a **DHS_BD_FRAME** is described in Section 7.0 on page 28. This structure is manipulated with the bulk data formatting functions described in Section 6.2 on page 19.

6.1.4 DHS_BD_GET_TYPE Indicates the format for retrieved data

This is an enumerated data type used to indicate how retrieved data should be formatted. Possible values for this type are:

DHS_BD_GT_FITS — Get the data as a FITS file.

DHS_BD_GT_FITS_HEADER — Get the FITS header only.

DHS_BD_GT_RAW — Get the data as stored without any conversion at all.

DHS_BD_GT_RAW_ASIS — For internal use only. Return a raw file as it was stored. (This is intended for DHS internal use only.)

DHS_BD_GT_FITS_ASIS — For internal use only. Return a FITS file as it was stored. (This is intended for DHS internal use only.)

DHS_BD_GT_UNKNOWN — The format of the data is unknown or not applicable.

6.1.5 DHS_BD_LIFETIME Lifetime of the data associated with a dataset name

This is an enumerated data type which indicates the lifetime of the data associated with a dataset name. Possible values for this data type are:

DHS_BD_LT_PERMANENT — Data will be saved to the DHS permanent store. Most “normal observations” will be permanent.

DHS_BD_LT_TEMPORARY — Data will not be written to the Gemini permanent store, but will be available for retrieval by other Gemini systems and for use by the DHS data processing systems. Temporary data should be deleted with function **dhsBdDelete** when it is no longer required, and will be removed automatically whenever the DHS data server is reset. This feature is included to allow Gemini systems to send bulk data for use by synchronous data processing routines and retrieve the results, without having to make the data a part of the Gemini permanent store.

DHS_BD_LT_TRANSIENT — Data only exists to be displayed by Quick Look. The data is not written to the permanent store, cannot be retrieved by any Gemini system and cannot be delivered to the observer.

6.1.6 DHS_BD_PUT_TYPE Type of data being sent

This is an enumerated data type which indicates the type of data being sent with the **dhsBdPut** function. Possible values for this data type are:

DHS_BD_PT_DS — The data is a **DHS_BD_DATASET** structure.

DHS_BD_PT_DS_QL — The data is a **DHS_BD_DATASET** structure but the data should only be sent to the Quick Look display, and will not become part of the permanent data.

DHS_BD_PT_FITS — The data is a FITS data buffer.

DHS_BD_PT_FITS_UNIQUE — Data is a FITS data buffer. Add a counter to the end of the dataset name. (This is intended for DHS internal use only.)

DHS_BD_PT_RAW — The data is a binary data buffer.

DHS_BD_PT_RAW_UNIQUE — Data is a binary data buffer. Add a counter to the end of the dataset name. (This is intended for DHS internal use only.)

DHS_BD_PT_SDS — The data is an exported SDS buffer (this should only used internally by the DHS).

DHS_BD_PT_UNKOWN — Data has an unknown data type.

6.1.7 DHS_CB_TYPE — Type of callback function

This is an enumerated type used to indicate the type of callback function being set in function **dhsCallbackSet**. The possible values of **DHS_CB_TYPE** that apply to the ICD 3 interface (values that only apply to the ICD 1c interface are omitted) are:

DHS_CBT_ERROR — DHS library error callback function.

DHS_CBT_CONNECT — The connection callback function.

DHS_CBT_GET — Function executed on a client system when requested data is available.

DHS_CBT_PUT — Function executed on a client system when data sent to a Data Server has been received.

DHS_CBT_SERVER_GET — Function executed on a server system when a client requests the data associated with a dataset name.

DHS_CBT_SERVER_PUT — Function executed on a server system when a server sends data to the server.

6.1.8 DHS_STATUS — Status value returned by functions in the DHS library

This is an enumerated type indicating the return status of functions in the DHS library. The function return values that apply to ICD 3 are listed in Table 4 on page 18. Status codes that are prefixed by "DHS_E_" are error or severe status codes, other codes are prefixed by "DHS_S_" and are warning or information codes. A complete list of return values is included in [6].

TABLE 4. ICD 3 Function return status values.

Status Symbol	Error level	Description
DHS_S_SUCCESS	**DHS_EL_INFO**	Normal completion of the function.
DHS_E_AVLIST_ARRAY	**DHS_EL_ERROR**	Attempted to add an array of AV lists as an attribute in an AV list.
DHS_E_INIT	**DHS_EL_SEVERE**	An attempt was made to use the DHS library before calling function **dhsInit**.
DHS_E_MEMORY	**DHS_EL_SEVERE**	A memory allocation failure has occurred.
DHS_E_NO_ATTRIB	**DHS_EL_ERROR**	A required attribute was not found in a command.
DHS_E_NO_LABEL	**DHS_EL_ERROR**	A requested dataset could not be found.
DHS_E_NOT_AVLIST	**DHS_EL_ERROR**	The specified SDS structure was not an AV list.
DHS_E_NULLVALUE	**DHS_EL_ERROR**	An attribute had a null value where a null value is not permitted.

TABLE 4. ICD 3 Function return status values. (Continued)

Status Symbol	Error level	Description
DHS_S_SHUTDOWN	**DHS_EL_INFO**	The command server wants to shut down. The client receiving this message should close its connection to the command server.
DHS_E_CON_LOST	**DHS_EL_ERROR**	A connection to a server or client was lost unexpectedly. The connection is no longer valid.
DHS_E_EL_RUNNING	**DHS_EL_ERROR**	An attempt was made to run the event loop when the event loop was already running.
DHS_E_SDS	**DHS_EL_SEVERE**	An error occurred in an SDS routine.
DHS_E_TYPE	**DHS_EL_TYPE**	An unknown data type was specified.

6.2 Bulk data formatting functions

This following functions are used to create populate and interpret **DHS_BD_DATASET** structures. None of these functions access the Data Server, they only manipulate **DHS_BD_DATASET** structures on the local system.

6.2.1 dhsBdDsNew — Create a new **DHS_BD_DATASET** structure

```
DHS_BD_DATASET          dhsBdDsNew
(
    DHS_STATUS          *status        /* out   : Function return status.               */
)
```

This function creates a new, empty **DHS_BD_DATASET** structure and returns a pointer to it. The DHS_BD_DATASET structure can be manipulated with the functions **dhsBdFree**, **dhsBdFrameNew**, **dhsAttribAdd**, **dhsAttribInfo**, and **dhsAttribFind**.

6.2.2 dhsBdDsFree — Free the memory allocated to a dataset structure

```
void                    dhsBdFree
(
    DHS_BD_DATASET      dataset        /* in    : The dataset structure to free.        */
    DHS_STATUS          *status        /* out   : Function return status.               */
)
```

This function frees all memory allocated to the **DHS_BD_DATASET** structure referred to by the *dataset* parameter.

6.2.3 dhsBdFrameNew — Add a new frame to a dataset structure

```
DHS_BD_FRAME            dhsBdFrameNew
(
    object-type         object,        /* in    : The object for the frame.             */
    char                *name,         /* in    : The name of the frame.                */
    int                 index,         /* in    : The index number of the frame.        */
    DHS_DATA_TYPE       dataType,      /* in    : The type of data in the frame.        */
    int                 ndims,         /* in    : Number of dimensions of the data array.*/
    unsigned long       *dims,         /* in    : Size of each axis of the data array.   */
    void                **dataPointer, /* out   : Pointer to the data array.            */
    DHS_STATUS          *status        /* out   : The function return status.           */
)
```

This function adds a new empty frame to the frame or dataset specified in the *object* parameter. The *object* parameter must by of type **DHS_BD_DATASET** or **DHS_BD_FRAME**. The new frame is returned. The *index* parameter allows specific frames to be created without creating the whole frame structure. (See Section 8.0 on page 35 for a description of frame indexes.) Data for the frame should be written into the memory buffer returned in the *dataPointer* parameter. If the frame will not contain a data array, a data type of **DHS_DT_NONE** can be passed in the *dataType* parameter, the *ndims* and *dims* parameters will be ignored, and NULL will be returned in the *dataPointer* parameter. The frame structure can be manipulated with the functions **dhsBdFrameFind**, **dhsBdFrameInfo**, **dhsAttribAdd**, **dhsAttribInfo**, and **dhsAttribFind.**

6.2.4 dhsBdFrameInfo Get information about a frame

```
void                  dhsBdFrameInfo
(
    DHS_BD_FRAME       frame,         /* in    : The frame to get.                   */
    char               **name,        /* out   : The name of the frame.              */
    DHS_DATA_TYPE      *type,         /* out   : The type of the data.               */
    int                *naxis,        /* out   : The number of axes of the data.     */
    unsigned long      *naxes,        /* out   : The size of each axes.              */
    void               **value,       /* out   : A pointer to the data array.        */
    DHS_STATUS         *status        /* out   : Function return status.             */
)
```

This function returns information about a frame. The *frame* parameter indicates which frame to query. The *naxes* parameter must be an array of 7 integers to allow it to contain the maximum number of dimensions supported by SDS.

6.2.5 dhsBdFrameFind Find a frame by name

```
DHS_BD_FRAME          dhsBdFrameFind
(
    object-type        object,        /* in    : The object to search.               */
    char               *name,         /* in    : The name to look for.               */
    DHS_STATUS         *status        /* in    : The function return status.         */
)
```

This function searches a dataset or frame structure looking for a frame whose name (as specified when the frame was created with **dhsBdFrameNew**) is the same as the name specified in the *name* parameter, and returns the index number of the frame. The *object* parameter must be of type **DHS_BD_DATASET** or **DHS_BD_FRAME**. If no frame is found corresponding to the *name* parameter, the function will return status **DHS_S_NO_FRAME**. Information about the returned frame can be retrieved with function **dhsBdFrameInfo**.

6.2.6 dhsBdFrameIndex Find a frame based on its position in the dataset structure

```
DHS_BD_FRAME          dhsBdFrameIndex
(
    object-type        object,        /* in    : The object to search.               */
    int                index,         /* in    : Position of the frame in the object.   */
    DHS_STATUS         *status        /* out   : Function return status.             */
)
```

Find a frame based on its position in an object. The *index* parameter indicates the position of the frame within the object, and should be the same value that was passed in the *index* parameter of the **dhsBdFrameNew** function. If no frame is found corresponding to the *name* parameter, the function will return status **DHS_S_NO_FRAME**. Information about the returned frame can be retrieved with function **dhsBdFrameInfo**.

6.2.7 dhsBdAttribAdd Add an attribute to a frame or dataset structure

```
void                  dhsBdAttribAdd
(
    object-type        object,        /* in    : The object to add data to.          */
    char               *name,         /* in    : The name of the attribute.          */
    DHS_DATA_TYPE      attType,       /* in    : The type of the attribute.          */
    int                ndims,         /* in    : Number of dimensions of attribute.  */
    unsigned long      *dims,         /* in    : The size of each dimension.         */
    ?                  value,         /* in    : The value of the attribute or a pointer*/
                                      /*         to the attribute array.             */
    DHS_STATUS         *status        /* out   : The function return status.         */
)
```

This function allows an new attribute to be added to a data object. The *object* parameter must be of type **DHS_BD_DATASET** or **DHS_BD_FRAME**. The type of the *value* parameter depends on the *attType* and *ndims* parameters. If the *ndims* parameter is 0, the value is assumed to be a scalar value, the *dims* parameters is ignored, and the *value* parameter should be the actual data item. If the *ndims* parameter is greater than zero, the size of each dimension is given in the *dims* parameter and the *value* parameter should be a pointer to the data item. See Section 10.2 on page 39 for examples of the use of this function.

6.2.8 dhsBdAttribDelete — Delete an attribute and its value from a dataset or frame

```
void                    dhsBdAttribDelete
(
    object-type         object,        /* in    : The object to be changed.              */
    const char          *attName,      /* in    : The name of the attribute to delete.   */
    DHS_STATUS          *status        /* mod   : Function return status.                */
)
```

This function attempts delete an attribute with the name *attName* from a dataset or frame. The *object* parameter must be of type **DHS_BD_DATASET** or **DHS_BD_FRAME**. If the attribute cannot be found, a status of DHS_S_NO_ATTRIB is returned. Note that this function will change the index numbers used by function **dhsBdAttribIndex** for all attributes in the object where the index is greater than the index of the deleted attribute.

6.2.9 dhsBdAttribInfo — Get information about an attribute

```
void                    dhsBdAttribInfo
(
    DHS_AV_ID           attrib,        /* in    : Attribute to check.                    */
    char                **name,        /* out   : The name of the attribute.             */
    DHS_DATA_TYPE       *type,         /* out   : The type of the attribute.             */
    int                 *ndims,        /* out   : Number of dimensions of the attribute. */
    int                 *dims,         /* out   : Array to contain the size of each      */
                                       /*          dimension.                            */
    void                **value,       /* out   : Pointer to the value.                  */
    DHS_STATUS          *status        /* out   : Function return status.                */
)
```

This command gets information about an attribute from an attribute value list. The *attrib* parameter indicates which attribute to query. The *dims* parameter indicates the dimensions of the attribute value. The *dims* parameter must be an array of 7 integers to allow it to contain the maximum number of dimensions supported by SDS.

6.2.10 dhsBdAttribFind — Find an attribute by name

```
DHS_AV_ID               dhsBdAttribFind
(
    object-type         object,        /* in    : The object to search.                  */
    char                *name,         /* in    : The attribute to find.                 */
    DHS_STATUS          *status        /* out   : The function return status.            */
)
```

This function locates an attribute by name and returns the attributes item number. The *object* parameter indicates the object to search, and must be of type **DHS_BD_DATASET** or **DHS_BD_FRAME**. An application can use function **dhsBdAttribInfo** to get more information about the attribute.

6.2.11 dhsBdAttribIndex — Find an attribute based on its position in a frame or dataset

```
DHS_AV_ID               dhsBdAttribIndex
(
    object-type         object,        /* in    : The object to search.                  */
    int                 index,         /* in    : The position in the object.            */
    DHS_STATUS          *status        /* mod   : The function return status.            */
)
```

This function finds an attribute based on position in a frame or dataset. The *index* parameter indicates where the attribute is in the frame or dataset, where 0 indicates the first item in the list. If no attribute can be found matching the index number, the function will return with status

DHS_S_NO_ATTRIB. An application can use function **dhsBdAttribInfo** to get more information about the attribute.

6.2.12 dhsBdDsAccess — Access an dataset in a memory buffer

```
DHS_BD_DATASET          dhsBdDsAccess
(
    const void          *buffer,       /* in    : The buffer to access.                 */
    DHS_STATUS          *status        /* mod   : Function return status.               */
)
```

This function makes a memory buffer created by the **dhsBdDsExport** function available as a read-only dataset. This is primarily a debugging/simulation support function. The AV list returned by this function should be freed with the **dhsBdDsFree** function when it is no longer needed. Note that this function creates a dataset that points to data in the buffer, so it is necessary to keep the buffer intact for as long as the dataset is needed. If a stand alone, writable dataset is required, a copy can be made with the **dhsBdDsCopy** function.

6.2.13 dhsBdDsCopy — Copy a dataset

```
DHS_BD_DATASET          dhsBdDsCopy
(
    DHS_BD_DATASET      dataset,       /* in    : The dataset to copy.                  */
    DHS_STATUS          *status        /* mod   : Function return status.               */
)
```

This function creates a new dataset containing a copy of the contents of an existing dataset. This function should be used sparingly, especially on large datasets, since it isn't very efficient. The new dataset must be freed with the **dhsBdDsFree** function when it is no longer needed.

6.2.14 dhsBdDsExport — Export a dataset to a memory buffer

```
void                    dhsBdDsExport
(
    DHS_BD_DATASET      dataset,       /* in    : The dataset to export.                */
    void                *buffer,       /* mod   : The buffer to receive the AV list.    */
    unsigned int        bufSize,       /* in    : The size of buffer.                   */
    DHS_STATUS          *status        /* mod   : The function return status.
)
```

This function exports a dataset to a memory buffer. The **dhsBdDsAccess** function can then be used to make the buffer usable as a dataset again (usually after writing it to disk, or sending it as a message, or some other thing that can only be done to a contiguous memory buffer). The buffer allocated before calling this function, and must be large enough to contain the exported structure. The expected size of the exported structure can be determined with the **dhsBdDsSize** function. This is primarily a debugging/simulation support function.

6.2.15 dhsBdDsPrint — Print a dataset to standard out

```
void                    dhsBdDsPrint
(
    DHS_BD_DATASET      dataset,       /* in    : The dataset to print.                 */
    DHS_STATUS          *status        /* mod   : The function return status.           */
)
```

This function prints the contents of the dataset to standard output in a readable form. Because of its limitations, this function is only useful for debugging.

6.2.16 dhsBdDsSize — Return the size of a dataset

```
unsigned long        dhsBdDsSize
(
    DHS_BD_DATASET     dataset,       /* in    : The dataset to size.                   */
    DHS_STATUS         *status        /* mod   : Function return status.                */
)
```

This function determines the amount of space a dataset would occupy if it were exported to a memory buffer. This function is used prior to calling **dhsBdDsExport** to determine how big the buffer must be.

6.3 Client bulk data transfer functions

The following functions will be used primarily by client systems for transferring datasets after they have been created with the data formatting functions.

6.3.1 dhsBdCtl — Send a control command to the Data Server

```
void                 dhsBdCtl
(
    DHS_CONNECT        connect,       /* in    : The data server connection.            */
    DHS_BD_CTL         ctl,           /* in    : The type of control function.          */
    ...,                              /* in    : Optional parameters for the ctl.       */
    DHS_STATUS         *status        /* out   : The function return status.            */
)
```

This function sends a control signal to the data server. Valid control signals are:

DHS_BD_CTL_ABORT — Abort the data transfer and discard all data for the dataset. The optional parameter for this control function is a string containing the dataset name.

DHS_BD_CTL_RESET — Reset the data array and all attributes to their un-set state. The optional parameter for this control function is a string containing the dataset name (see Section 10.7 on page 45).

DHS_BD_CTL_LIFETIME — Change the lifetime assigned to a dataset. The optional parameters for this control function are a string containing the dataset name, and a **DHS_BD_LIFETIME** value.

DHS_BD_CTL_CONTRIB — Change the list of contributors to a dataset. The optional parameters for this control function are a string containing the dataset name, an integer containing the size of the contributor array, and an array of strings containing the new contributor list.

DHS_BD_CTL_QLSTREAM — Change the list of Quick Look streams associated with a dataset. The optional parameters for this control function are a string containing the dataset name, an integer containing the size of the Quick Look stream list, and an array of strings containing the new list of Quick Look streams.

DHS_BD_CTL_GETNAME — Get a new unique name from the Data Server. (See Section 8.0 on page 35). The optional parameter for this control function is a pointer to a character pointer (char **) which will returned pointing to the new unique name.

These actions appear as ICD 1c commands on the Data Server. The command ICD 1c callback will be executed and the command name passed to the callback will be “bdCtl”. Input parameters for the control function are be passed as attributes in the commands attribute value list, and the control type is passed as an integer attribute named “ctl”, which will contain one of the values defined by the **DHS_BD_CTL** type. Output parameters for the control function will be returned in a response attribute value list.

The attribute names used when passing parameters in commands are:

dataset name — Passed as attribute **datasetName**.

lifetime — Passed as attribute **lifeTime**.

quick look stream names — Passed as attribute **nameArray**.

contributor names — passed as attribute **nameArray**.

data label — Returned as attribute **dataLabel**.

6.3.2 dhsBdName — Retrieve a new unique name from the Data Server

```
char                *dhsBdName
(
    DHS_CONNNECT       connect,       /* in    : The data server connection.          */
    DHS_STATUS         *status        /* mod   : The function return status.          */
)
```

This function requests a new unique name from the Data Server and returns a pointer to the new name. This function is implemented as a call to the **dhsBdCtl** function.

6.3.3 dhsBdGet — Get the data for a dataset name

```
DHS_TAG             dhsBdGet
(
    DHS_CONNECT        connect,       /* in    : The Data Server connection.          */
    char               *datasetName,  /* in    : The dataset to get.                  */
    DHS_BD_GET_TYPE    getType,       /* in    : Format for returned data.            */
    void               *userData,     /* in    : User data pointer.                   */
    DHS_STATUS         *status        /* mod   : The function return status.          */
)
```

This function gets data associated with a dataset name from the Data Server. The dataset must be complete and available to the Data Server before this function can be called successfully. The *getType* parameter indicates what form the retrieved data should take, and is described in Section 6.1.4 on page 17. This verifies the data is available, and if it is, returns without waiting for the data to be retrieved, returning a **DHS_TAG** which can be queried with the **dhsStatus** and **dhsWait** functions. It is the responsibility of the calling program to free the returned **DHS_TAG** with the **dhsTagFree** function after the request has been completed.

The *userData* parameter can be set by an application to associate a pointer with the **DHS_TAG** (NULL may be specified if user data is not being used), as described in [6]. If the DHS_CBT_GET callback function has been set, when a response is received from a server for the data request, or when the transfer is complete, the callback function will be executed (see Section 6.5.1 on page 27). When the data request is complete, the returned data can be accessed with the data pointer in the data get callback function.

The final command status of the request will be **DHS_CS_DONE** if the data could be found and retrieved, or **DHS_CS_ERROR** if the data could not be retrieved. The status string will contain a string describing details the success or failure of the operation.

The status return value from this function only indicates the success of this function, not the status of the resulting command. If the **dhsBdGet** function succeeds, then the status of the command can be queried by passing the returned **DHS_TAG** to the **dhsStatus** function. If the **dhsBdGet** function fails then no command is created, the returned value will be **DHS_TAG_NULL**, and there is nothing for **dhsStatus** to query.

6.3.4 dhsBdPut Send a dataset or chunk of a dataset to the Data Server

```
DHS_TAG             dhsBdPut
(
    DHS_CONNECT         connect,        /* in    : Connection to the Data Server          */
    char                *datasetName,   /* in    : Dataset name for the data.             */
    DHS_BD_PUT_TYPE     putType,        /* in    : Type of data being sent.               */
    DHS_BOOLEAN         last,           /* in    : Is this the last data sent?            */
    ...,                                /*       : Optional parameters.                   */
    void                *userData,      /* in    : User data pointer.                     */
    DHS_STATUS          *status         /* mod   : The function return status.            */
)
```

This function is used to send bulk data to the Data Server. The function returns immediately, returning a **DHS_TAG** which can be used to identify the request. It is the responsibility of the calling program to free the **DHS_TAG** with the **dhsTagFree** function after the request has been completed. The *userData* parameter can be set by an application to associate a pointer with the **DHS_TAG** (NULL may be specified if user data is not being used), as described in [6]. The **dhsStatus** and **dhsWait** functions can be used to wait for completion of the data transfer. If the **DHS_CBT_PUT** callback function has been set, the callback function will be executed when a response is received from the receiver, or when the transfer is complete (see Section 6.5.2 on page 27).

When the *last* parameter is set to **DHS_TRUE**, it indicates that no more data will be sent for the dataset by this Gemini system, otherwise the *last* parameter should be set to **DHS_FALSE**. Each client sending data to the Data Server must set the last flag to true for the last chunk of data sent for a dataset.

The optional parameters depend on the *putType* parameter. If the *putType* parameter is **DHS_BD_PT_DS** or **DHS_BD_PT_DS_QL**, then the only optional argument is a **DHS_BD_DATASET**. If the *putType* parameter is anything else, the optional arguments are a void pointer to the data buffer, and the size of the data buffer.

The command status of the request (as reported in the **DHS_CBT_PUT** callback function, or with the **dhsStatus** function) will be set to **DHS_CS_DONE** if the put was processed in a completely normal way. If any deviations from normal processing occurred, the returned status will be **DHS_CS_ERROR**. Note that an error could be relatively minor condition, for instance an unrecognized attribute in a dataset, a severe error which resulted in the data not being saved by the server at all, or anything in between. The status string will contain a string describing details the success or failure of the operation.

The status return value from this function only indicates the success of this function, not the status of the resulting command. If the **dhsBdPut** function succeeds, then the status of the command can be queried by passing the returned **DHS_TAG** to the **dhsStatus** function. If the **dhsBdPut** function fails then no command is created, the returned value will be **DHS_TAG_NULL**, and there is nothing for **dhsStatus** to query.

6.3.5 dhsBdDelete Delete a dataset from a Data Server

```
void                dhsBdDelete
(
    DHS_CONNECT         connect,       /* in    : Connection to the server.            */
    char                *datasetName,  /* in    : Dataset to delete.                   */
    DHS_STATUS          *status        /* mod   : Function return status.              */
)
```

This function deletes the dataset indicated by parameter datasetName from the Data Server. It is only possible to delete datasets whose lifetime is **DHS_BD_LT_TEMPORARY** or where data has been partially or not been received.

The server will receive this function call as an ICD 1c command with a command name of "**bdDelete**", and with the name of the dataset in a "**datasetName**" attribute.

6.4 Server bulk data transfer functions

These functions are used by a server to respond to client request.

6.4.1 dhsBdResponse Send a response containing bulk data to a client

```
void                dhsBdResponse
    DHS_CONNECT         connection,    /* in    : Connection to the client.            */
    DHS_TAG             tag,           /* in    : Clients DHS_TAG.                     */
    DHS_CMD_STATUS      *status,       /* in    : New status of the request.           */
    char                *statusString, /* in    : New status description.              */
    DHS_AV_LIST         avList,        /* in    : Attributes to return to client.      */
    DHS_BD_GET_TYPE     getType,       /* in    : Type of data being returned.         */
    void                *data,         /* in    : Pointer to the data to return.       */
    unsigned long       length,        /* in    : Length of the data to return.        */
    DHS_STATUS          *status        /* mod   : Function return status.              */
)
```

This function is used to send replies which include bulk data. The *avList* parameter is optional and can be set to DHS_AV_LIST_NULL if it is not used. The statusString parameter is optional and can be set to NULL if it is not used.

Note that the **dhsBdGet** function executed by the client will not return until the server changes the command state with this function. Because of this, a server should use this function to change the command state to **DHS_CS_BUSY** soon after it starts processing a request. The possible values for the command status are listed in [6].

6.5 User supplied functions

These functions are supplied by applications using the ICD 3 interface, and are set using the **dhsCallbackSet** function. The names of these functions are not significant. The processing of new events will not proceed until these functions return, so it may be necessary to create a thread for time consuming actions.

6.5.1 Data get callback

Function called on the client when requested data is available

```
void                  data_get_callback
(
    DHS_CONNECT       connect,       /* in   : The connection to the server.       */
    DHS_TAG           tag,           /* in   : Tag associated with the get.        */
    char              *datasetName,  /* in   : The requested dataset name.         */
    DHS_BD_GET_TYPE   type,          /* in   : The format of the data.             */
    DHS_CMD_STATUS    status,        /* in   : The command return status.          */
    char              *string,       /* in   : The status message string.          */
    DHS_AV_LIST       avList,        /* in   : The response AV list.               */
    void              *data,         /* in   : A pointer to the requested data.    */
    unsigned long     length,        /* in   : The length of the data buffer.      */
    void              *userData      /* in   : User data pointer.                  */
)
```

This function is called when data requested with function **dhsBdGet** is available on the client. The data will be stored in the IMP shared memory when this function is executed, and will not be released until the callback returns. This callback function is set by calling **dhsCallbackSet** with the *cbType* parameter set to **DHS_CBT_GET**.

The status parameter will contain a status value (possible values are listed in [6]) indicating whether the request succeeded or failed. The string parameter will contain details of the success or failure of the operation. The application should be prepared for the *string* parameter to be NULL, for the *avList* parameter to be DHS_AV_LIST_NULL, for the *data* parameter to be NULL, or for the returned data type to differ from the requested data type.

6.5.2 Data put callback

Function called on the client when data storage is complete

```
void                  data_put_callback
(
    DHS_CONNECT       connect,       /* in   : The connection to the server.       */
    DHS_TAG           tag,           /* in   : Tag associated with the put.        */
    DHS_CMD_STATUS    status,        /* in   : The command return status.          */
    char              *string,       /* in   : The status message string.          */
    char              *datasetName,  /* in   : Dataset name for the put.           */
    void              *userData      /* in   : User data pointer.                  */
)
```

This function is called when data sent with the **dhsBdPut** function has been safely stored by the data server. This callback function is set by calling **dhsCallbackSet** with the *cbType* parameter set to **DHS_CBT_PUT**.

The status parameter will contain a status value indicating if the request succeed or failed. The string parameter will contain details of the success or failure of the operation. If an attribute value list was returned with the response, the list can be retrieved with the ICD 1c **dhsResponseGet** function.

6.5.3 Server get callback

Function called on the server system when data is requested

```
void                  server_get_callback
(
    DHS_CONNECT       connect,       /* in   : The connection to the client.       */
    DHS_TAG           tag,           /* in   : Client request tag.                 */
    char              *datasetName,  /* in   : Requested dataset name.             */
    DHS_DB_GET_TYPE   type           /* in   : The type of data requested.         */
)
```

This function is called on a server system in response to a client calling the **dhsBdGet** function. This callback function should attempt to retrieve the requested data in the specified form, and return it to the client system using the **dhsBdResponse** function. This callback function is set by calling **dhsCallbackSet** with the *cbType* parameter set to **DHS_CBT_SERVER_GET**.

6.5.4 Server put callback

Function called on the server system when data is sent

```
void                    server_put_callback
(
    DHS_CONNECT         connect,       /* in    : The connection to the client.         */
    DHS_TAG             tag,           /* in    : Client request tag.                   */
    char                *datasetName,  /* in    : The dataset name for the data.        */
    DHS_BD_PUT_TYPE     type,          /* in    : The type of data being put.           */
    void                *data,         /* in    : Pointer to the data.                  */
    unsigned long       length         /* in    : Length of the data buffer.            */
)
```

This function is called on a server system in response to a client calling the **dhsBdPut** function. This function should store the data and then use function **dhsCmdResponse** to set the request status to **CS_CS_DONE**. This callback function is set by calling **dhsCallbackSet** with the *cbType* parameter set to **DHS_CBT_SERVER_PUT**.

If the type parameter is **DHS_BD_PT_DS** or **DHS_BD_PT_DS_QL**, then the data can be made accessible as a dataset structure using the **dhsBdDsAccess** function. For any other data type, the data buffer can be used directly. If a new thread is created to handle the processing for this callback function, a copy of the data buffer must be made, since the buffer passed in is unsafe to use after this routine returns.

7.0 Data Structures

This section describes the standard data structures containing a dataset, which may be transmitted with the functions described in Section 6.0 on page 16. The data structures are hierarchical, with the more complex structures being built from the basic structures. Such hierarchical structures are necessary to support the many kinds of data which the Gemini instruments may generate.

Note that this document is not designed to constrain the types of bulk data transmitted between the principal systems; only define a standard way of describing that data. Only a few of the most common data structures are presented here. New structures can be added to the list as required, but *only* with the consent of the Gemini project office.

These data structures are designed to allow for the transmission of arbitrarily complex data, however the data structures used by instrument system should be kept as simple as possible to allow them to be stored in standard FITS files.[1]

7.1 Primitive Data Types

The data types listed in Table 6 on page 25 of [6] are used to build up the data structures, as described in [14]. Arrays of any of these data types are also permitted. The DHS_INT32 type should be used for integer values unless there is a good reason not to (e.g. constraints of space). The bit and/or byte ordering of the data types is machine dependant. Any translation of machine dependant values will be done by the DHS library. The generic data type <numeric> is used in the following sections to represent any of the numerical data types. The generic data type <value> is used in the following sections to represent any data type.

1. In the example in Section 7.9.1 on page 33 the axisMap array is used to map the axis values into wavelength in microns. This array could be stored in the dataset's FITS file, however there is no standard way to store axisMap which would cause programs reading the FITS file to use it correctly.

7.2 Standards for Array Representation

All arrays in the data structures must be written in left to right, bottom to top, row major order (i.e. the natural order used in the C programming language). In all the following sections the “first element” of an array is assumed to be its bottom-left corner, and the axes of an array are assumed to go naturally from left to right and bottom to top. If any Gemini system generates data in a different order it must flip and/or rotate the data to meet this standard before transmitting it. See Figure 3 for details. In this figure the array elements are numbered in the order of increasing memory address. The first column and row are (1,1) in this example, and the first memory location is 1, but this need not be so. (The origin attribute of a frame header can specify a different origin for the array as described in Section 7.7.2 on page 31). Figure 4 on page 29 shows the mapping from an array onto a grid when there is no world coordinate information for a frame.

FIGURE 3. A 6x4 element 2-D array stored in left-right bottom-top row-major order.

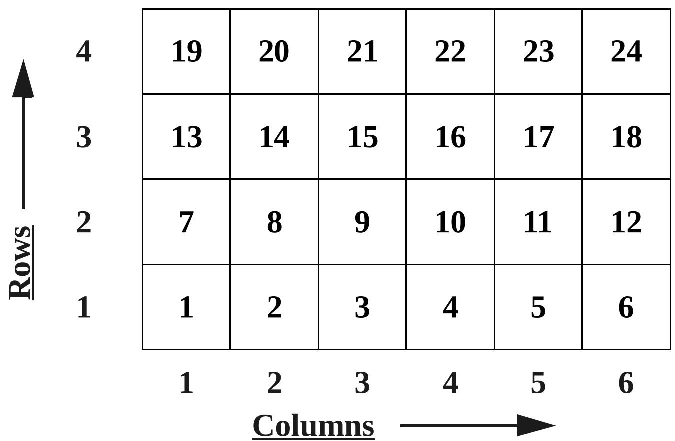
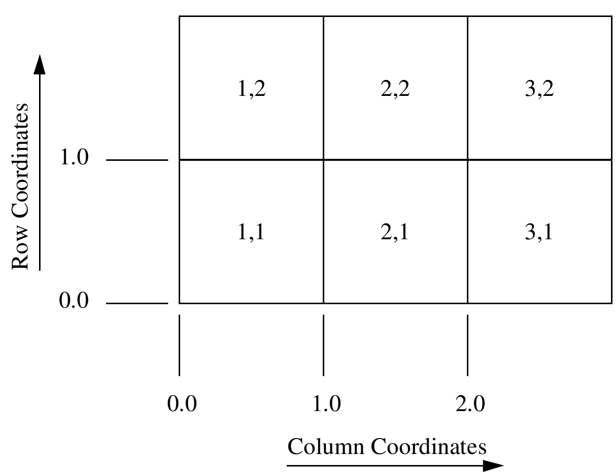


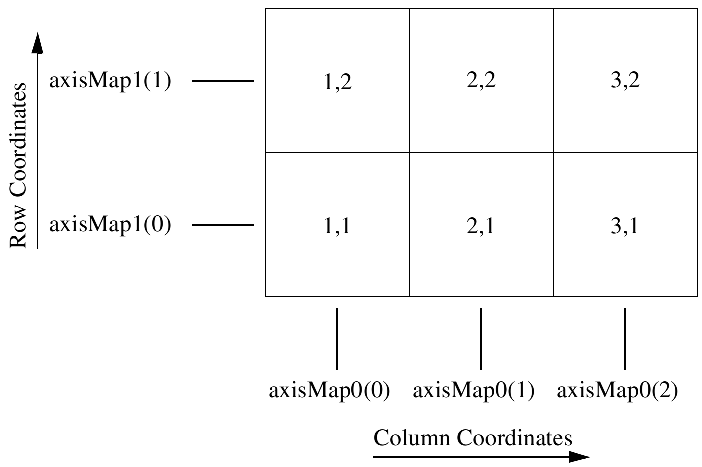
FIGURE 4. Mapping an array onto a grid without an axisMap structure.



7.3 World coordinate systems

A world coordinate system (WCS) can be mapped onto a data array using either an axisMap structure to map coordinates onto the axes of the data array, or using standard world coordinate system parameters stored as attributes in the frame header. The world coordinates should map to the centre of the pixels in the data array whichever type of WCS mapping is used. Figure 5 on page 30 shows the mapping from pixel to WCS when WCS information in an axisMap structure.

FIGURE 5. Mapping an array onto a grid with an axisMap structure.



7.4 Error and data quality

If pixel error and quality data are available, they should be stored as sub-frames of the data array, with types “Variance” and “Quality”. To ensure consistency of data between instruments, only these forms of error and quality information should be used.

7.5 Attribute/value pairs

Header attributes in the data structures must be defined in the DHS attribute/keyword dictionary. The values may be of any type, and may include arrays of values[1]. The attribute/keyword dictionary will contain the attribute properties as described in Section 3.1.1 on page 10.

7.6 Frames

Frames contain a frame identifier, a frame name, attribute/value data describing the frame, an optional single data array, and an optional set of sub-frames. The hierachical definition of frames allows related frames to be grouped in a logical way (see the example in Section 7.9.1 on page 33).

The frame identifier is an integer which is specified when a frame is created with the **dhsBdFrameNew** function, and which allows a frame to be located within the hierarchy.

1. The possible options for storing these arrays in FITS files are limited, but some of the options are indexed keywords and FITS table extensions.

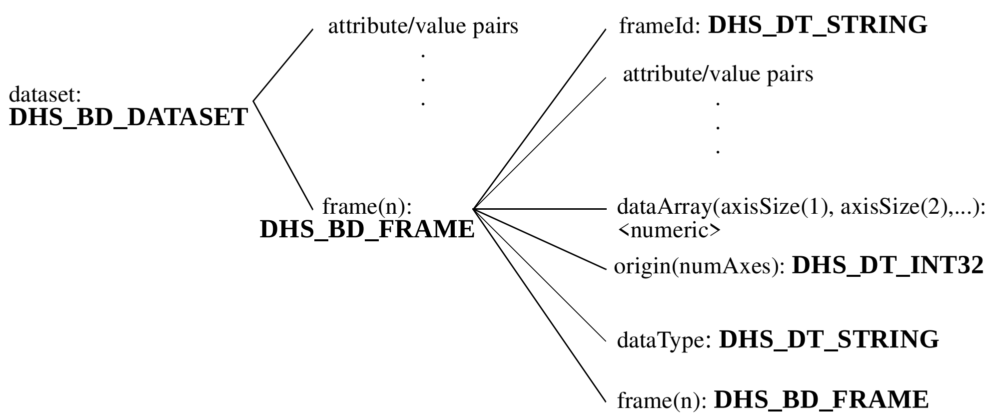
7.7 Data structure specifics

In all the following data structure examples, the first word on a line is the name of an item, the second word is the type of the item, and the rest of the line shows that item's value (if any) or the word "{structure}" if the item is a structure. Any items surrounded by parenthesis "()" are optional. The indentation shows the hierarchy. See Section 1.6 on page 6 for the conventions used. The dimensions of an array can be determined with DHS library routines, so array dimensions are not stored explicitly. The term "numAxes" refers to the number of axes of data in a frame, and the terms "naxis1, naxis2,..." are used to refer to the dimensions of each axis.

7.7.1 Dataset structure (DHS_BD_DATASET)

The **DHS_BD_DATASET** structure is used to contain raw or processed pixel and header data values. A graphical representation of the **DHS_BD_DATASET** structure is shown in Figure 6. If any associated data arrays are to be stored with the frame (for example pixel variance, data quality, number of exposures averaged for each pixel, etc.), then these data arrays should be stored as sub-frames within the raw frame, and the sub-frame header should describe the content of the data array (see the example in Section 7.9.1 on page 33). In the **DHS_BD_DATASET** structure *attribute name* would be an attribute name as described in the DHS attribute/keyword dictionary, and originally specified in the PDF files for each client system (mostly the instruments and OCS).

FIGURE 6. DATASET structure.



Following is an example of the contents of a **DHS_BD_DATASET** structure.

```
dataset                             DHS_BD_DATASET    {structure}
  (<attribute name>                 <value>           [value of the attribute])
                                    .
                                    .
                                    .
  (frame(n)                         DHS_BD_FRAME      {array of structures})
```

7.7.2 Frame data structure DHS_BD_FRAME

The **DHS_BD_FRAME** structure is used to store data from a frame or frame region. There is no distinction between the structure for frames and the structure for frame regions since a complete frame can be stored as a frame region. It is an error for a frame region to contain pixel values outside the array size specified in the frame header. In the following structure *attribute name* would be replaced by an attribute name defined in the DHS attribute/keyword dictionary.

If it is necessary to store data from exotic instruments, and with the consent of the Gemini project office, the “dataArray” member may be replaced with some other structure.

```
frameRegion                      DHS_DT_FRAME
  frameId                        DHS_DT_STRING    [This value indicates where the frame is
                                                  located in the hierarchy of frames. See
                                                  Section 8.0 on page 35 for more detail.]
  (<attribute name>              <value>          [The value of the attribute.])
  (dataArray(...)                <numeric>        [Contains the data of a frame or frame
                                                  region])
  (origin(numAxes)               DHS_DT_INT32     [The origin of each axis of a frame
                                                  region])
  (dataType                      DHS_DT_STRING    [The type of data contained in the data
                                                  array.])
  (frame(n)                      DHS_DT_FRAME     {array of structures}
```

7.8 Standard header attributes

Table 5 and Table 6 list some examples of standard attributes. The list of standard attributes will be expanded as the DHS is implemented. Non standard attributes will be defined by client systems in their PDF files.

TABLE 5. Dataset header standard attributes

Attribute name	Type	Optional	Description
object	**DHS_DT_STRING**	Yes	Name of object being observed
instrument	**DHS_DT_STRING**	No	Instrument name assigned by the Gemini project.
observer	**DHS_DT_STRING**	Yes	Name of the observer.
piName	**DHS_DT_STRING**	Yes	Name of the principal investigator
programId	**DHS_DT_STRING**	Yes	Gemini program identifier.
title	**DHS_DT_STRING**	Yes	Title for the dataset.
telescope	**DHS_DT_STRING**	No	Name of the telescope.
comment	**DHS_DT_STRING**	Yes	Optional observer comments

TABLE 6. Frame header standard attributes

Attribute name	Type	Optional	Description
frameId	**DHS_DT_STRING**	No	The frame id.
dataType	**DHS_DT_STRING**	No[a]	The type of data stored in the data array (“Photon count”, “variance”, etc.)
units	**DHS_DT_STRING**	Yes	Units of the data stored in the data array.
axisSize()	**DHS_DT_INT32**	Yes[a]	The dimensions of the data array.
axisLabel()	**DHS_DT_STRING**	Yes[b]	The label for each of the data array axes.
axisUnits	**DHS_DT_STRING**	Yes	The units for each of the data array axes.
axisMap<n>[c]	<numeric>	Yes[b]	An array used to map data array indexes into another coordinate system.

TABLE 6. Frame header standard attributes (Continued)

Attribute name	Type	Optional	Description
grey	<numeric>	Yes	The value indicating a data array element is undefined.
origin()	**DHS_DT_INT32**	Yes	If the frame contains a frame region, this array specifies the origin of the data array. This attribute will not become a part of the stored dataset.

a. Not required if the data array is not present.

b. If these attributes are not specified, the axes are assumed to be pixels.

c. axisMap0 applies to the first axis, axisMap1 applies to the second, etc.

7.9 Data Structure Examples

7.9.1 Example DATASET structure

The following data structure represents a 1024x256 pixel frame of M82 taken using a 2-D infra-red spectrometer. The header has been shortened, and in reality will contain many more items than shown here. Note that the data is accompanied by variance and quality arrays, as is common with infra-red observations. Many of the optional components are missing from this structure but the structure is still valid. It should be possible to generate, for example, a fully-labelled display with error bars from this structure. The code that could be used to create this structure is shown in Section 10.2 on page 39.

```
dataset                          DHS_BD_DATASET    {structure}
  object                         DHS_DT_STRING     “M82”
  telescope                      DHS_DT_STRING     “Gemini South”
  instrument                     DHS_DT_STRING     “2DIRS”
  observer                       DHS_DT_STRING     “Joe Astronomer”
  piName                         DHS_DT_STRING     “Joe Astronomer”
  programId                      DHS_DT_STRING     “Engineering”
  title                          DHS_DT_STRING     “Press release images”
  frame(1)                       DHS_BD_FRAME      {structure}
    frameId                      DHS_DT_STRING     “1”
    dataType                     DHS_DT_STRING     “Intensity”
    units                        DHS_DT_STRING     “photons”
    axisSize(1..2)               DHS_DT_INT32      1024, 256
    axisLabel(1..2)              DHS_DT_STRING     “Wavelength”, “Slit position”
    axisUnits(1..2)              DHS_DT_STRING     “Microns”, “Pixels”
    axisMap0(1..1024)            DHS_DT_FLOAT      (11.47488, 11.48512,...)
    dataArray(1..1024,1..256)    DHS_DT_FLOAT      555.5, 560.3, 580.4,...
    frame(1)                     DHS_BD_FRAME      {structure}
      frameId                    DHS_DT_STRING     “1.1”
      dataType                   DHS_DT_STRING     “Variance”
      units                      DHS_DT_STRING     “photons”
      axisSize(1..2)             DHS_DT_UINT32     1024, 256
      axisLabel(1..2)            DHS_DT_STRING     “Wavelength”, “Slit position”
      axisUnits(1..2)            DHS_DT_STRING     “Microns”, “Pixels”
      axisMap0(1..1024)          DHS_DT_FLOAT      (11.47488, 11.48512,...)
      dataArray(...)             DHS_DT_FLOAT      31.246, 35.214, 517.2, 12.11,...
    frame(2)                     DHS_BD_FRAME      {structure}
      frameId                    DHS_DT_STRING     “1.2”
      dataType                   DHS_DT_STRING     “Quality”
      axisSize(1..2)             DHS_DT_UINT32     1024, 256
      axisLabel(1..2)            DHS_DT_STRING     “Wavelength”, “Slit position”
      axisUnits(1..2)            DHS_DT_STRING     “Microns”, “Pixels”
      axisMap0(1..1024)          DHS_DT_FLOAT      (11.47488, 11.48512,...)
      dataArray(1..1024,1..256)  DHS_DT_INT8       0, 0, 0, 1, 0, 0, 7, 0, 0,...
```

7.9.2 Example SUBARRAY structure

This example describes a 512x512 pixel array sent to DHS in four separate parts (Header data from OCS, header data from ICS and the pixel array from ICS as two separate frame regions. Details of which system sends data to the DHS, and when the data is sent is not important to this interface. The important aspect of this example is the ability to construct a dataset from several groups, possibly sent by more than one Gemini system.

Header data. This structure contains the first group of header data. In practice the OCS will probably send header information to the ICS, which will add information of its own, and send the data on to DHS.

```
dataset                          DHS_BD_DATASET    {structure}
  object                         DHS_DT_STRING     “M82”
  instrument                     DHS_DT_STRING     “2DIRS”
  observer                       DHS_DT_STRING     “Joe Astronomer”
  piName                         DHS_DT_STRING     “Joe Astronomer”
  programId                      DHS_DT_STRING     “Engineering”
  title                          DHS_DT_STRING     “Press release images”
```

Frame header data from the ICS. This contains header data for a frame, generated by the ICS. In practice this data may be sent in several stages during the observe sequence.

```
frame                            DHS_BD_FRAME      {structure}
  frameId                        DHS_DT_STRING     “1”
  dataType                       DHS_DT_STRING     “Intensity”
  units                          DHS_DT_STRING     “photons”
  axisSize(1.2)                  DHS_DT_INT32      1024, 256
  axisLabel(1..2)                DHS_DT_STRING     “Wavelength”, “Slit position”
  axisUnits(1..2)                DHS_DT_STRING     “Microns”, “Pixels”
  axisMap0(axisSize(1..2))       DHS_DT_FLOAT      (11.47488, 11.48512,...)
  grey                           DHS_DT_FLOAT      0.0
```

First frame region from ICS.

```
frame                            DHS_BD_FRAME      {structure}
  frameId                        DHS_DT_STRING     “1”
  dataArray(512,256)             DHS_DT_FLOAT      555.5, 560.3, 580.4,...
  origin(2)                      DHS_DT_INT32      1,1
```

Second frame region from ICS.

```
frame                            DHS_BD_FRAME      {structure}
  frameId                        DHS_DT_STRING     “1”
  dataArray(512,256)             DHS_DT_FLOAT      530.33, 555.66, 554.7,...
  origin(2)                      DHS_DT_INT32      1,257
```

In this case the main array is completely covered by the individual arrays making it up, but this does not have to be the case. The **DHS_BD_FRAME** structure can be used to transmit data from oddly-shaped detectors (such as the T-shaped detector proposed for the echelle spectrograph). Any gaps between the subarrays are filled with the *grey* value. If no *grey* value is present the gaps between the subarrays should be regarded as bad pixels (and indicated as such in the QUALITY array). It is up to the application reading the data structure to deal with any overlaps.

An array may be sent to the DHS in several parts by sending it as several consecutive **DHS_BD_FRAME** structures which are only partially filled.

8.0 Data Labels

This section describes the structure of data names as agreed at the OCS/DHS Interface Meeting on June 12 and 13th 1996 (See [12] for a summary of the meeting.), and modified at a later meeting in Aug. 27, 1998.

The top level structure of a frame label is:

datasetName(:*frame*)

where the *datasetName* part of the label identifies the data for a specific dataset, and the optional *frame* part identifies a sub-frame or sub-tree of sub-frames (i.e. the specified frame and all frames contained within it).

The format of the datasetName part of the frame label is not significant to the ICD 3 interface, but is documented here since the users of the ICD 3 interface must be aware of it.

The structure of the *datasetName* is:

- ~~*unique-name.[[[.group-id].group-id] ...]*~~
- *unique-name[.dataStream][.index]*

where:

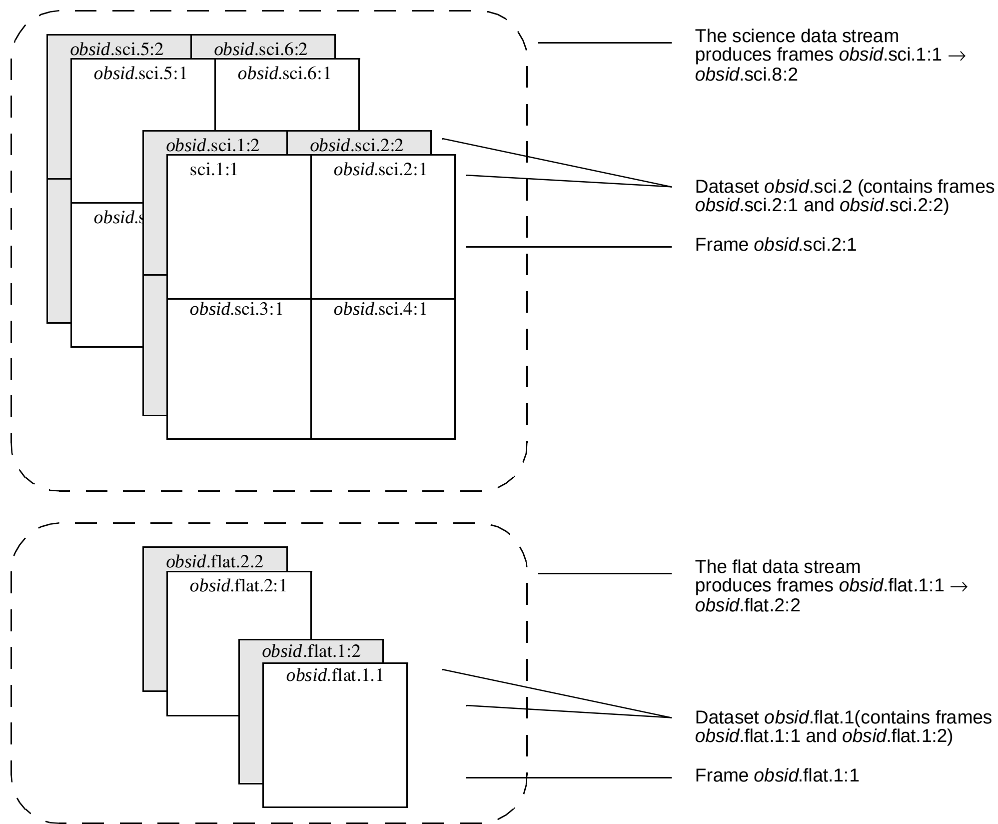
- *unique-name* is a unique string assigned by OCS for normal observations, or retrieved from the DHS using the **dhsBdName** function for engineering or other observations. When the *unique-name* is assigned by OCS, it can be used to associate datasets within a specific observation, and will be common to all datasets associated with that observation. The meaning of *unique-name* when assigned by DHS is less clear, but will probably be used to associate datasets collected for a single purpose.
- ~~*group-id* is an integer value which identifies the group to which the dataset belongs. This value identifies the datasets position at a given level in the hierarchy. The hierarchy can be of any depth~~
- *dataStream* is a string which indicates the type of data being collected. For normal observations, this value will be assigned by the OCS, and will indicate which of the observing program's Observe iterators caused the dataset to be created. For observations not under the control of the OCS, this label may be omitted, or assigned by the instrument.
- *index* is an integer counter used to create a unique dataset name for each unique name and data stream. The indexes must be reset to 1 for each new data label, and must increment by one for each dataset collected for a data stream.

~~The structure of the datasetName is designed to allow dataset names to indicate the structure of observations. An example of this is shown in Figure 7 on page 36. The example shown in the figure is a mosaic which has 2 frames per mosaic cell, a 2 by 2 mosaic, which is repeated two times, for a total of 16 frames in the observation. The order in which the frames are collected would be 1.1:1, 1.1:2, 1.2:1, 1.2:2, … 2.3:1, 2.3:2, 2.4:1, 2.4:2. The hierarchy is present to allow the datasets collected under the control of OCS to be organized within an observation, and may or may not be useful to structure engineering data originated by other principal systems.~~

~~It is up to client systems to assign the group-id values, and to ensure that the resulting dataset names are unique. In normal observations these values will be assigned by the OCS.~~

The structure of the dataset name is designed to allow dataset names to indicate the structure of observations. An example of this is shown in Figure 7 on page 36. The example shown in the figure is a mosaic which has 2 frames per mosaic cell, a 2 by 2 mosaic, which is repeated two times with a sky flat data set collected after each iteration of the mosaic, for a total of 20 frames in 10 datasets. The order in which the frames are collected would be sci.1:1, sci.1:2, ..., sci.4:1, sci.4:2, flat.1:1, flat.1:2, sci.5:1, sci.5:2, ... sci.8:1, sci.8:2, flat.2:1, flat.2:2. The structure is present to allow the datasets collected under the control of OCS to be organized within an observation, and may or may not be useful to structure engineering data originated by other principal systems.

FIGURE 7. Mosaic observation structure example



It is up to client systems to assign the *dataStream* and *index* values, and to ensure that the resulting dataset names are unique. In normal observations these values will be assigned by the OCS.

The structure of the *frame* part of the frame label is:

- *frame-id*(*.sub-frame-id*(*.sub-frame-id* …))

where the *frame* and *sub-frame* values are integers representing the index of each frame within the dataset or frame. The frame indexes are assigned by client systems when frames are created with the **dhsBdFrameNew** function.

~~An example of a complete frame label is:~~

- ~~*OCS-observation-id*.2.3:3.2.0~~

~~which identifies a frame in observation "*OCS-observation-id*". (The format of the "*OCS-observation-id*" isn't critical to this interface.) The group numbers are 2 and 3. The *frame* part of the frame label (3.2.0) identifies the first sub-frame (0) of the third sub-frame (2) of the fourth frame (3) of the dataset.~~

An example of a complete frame label is:

- *OCS-observation-id*.sci.10:3.2.1

which identifies a frame in observation "*OCS-observation-id*". (The format of the "*OCS-observation-id*" is irrelevant to this interface.) The data stream is "sci", and the index number is 10. The *frame* part of the frame label (3.2.1) identifies the first sub-frame of the second sub-frame of the third frame of the dataset.

9.0 Quick Look Streams

Quick look streams are used to identify which Quick Look displays should receive data for a dataset or dataset chunk. Quick look streams are identified by a string which is assigned by the client systems, and generally should be identified in the PDF file produced by the client systems to describe the data being transmitted to the DHS. (Identifying Quick Look streams in the PDF file allows the Quick Look Display to present a list of available Quick Look streams to the user. If a stream is not identified in the PDF file, the user of the Quick Look display will still have the option of entering the Quick Look stream name explicitly.)

Quick Look stream identifiers may be any ASCII string, but if there is an underlying structure to a group of streams, then that structure should be reflected in the stream names. An example of Quick Look streams with an underlying structure would be an instrument which sends science data from a science CCD, science data from a WFS, engineering data from the science CCD, and engineering data from the WFS. These streams could be named: "*inst*.CCD", "*inst*.WFS", "*inst*.eng.CCD", "*inst*.eng.WFS".

10.0 Example Code

The following sections illustrate the use of the bulk data interface. Error checking has been minimized in the examples to simplify them.

10.1 Initialization of a client application

The following example shows the initial set-up steps that should be done by a client system. The setting of the **DHS_CBT_PUT** callback function is not required unless the client needs callbacks after sending data to a server. The setting of the **DHS_CBT_GET** callback function is not required unless the client needs callback after retrieving data from a server.

Note that if the application is also using the ICD 1c interface to access a DHS command server, much of the initialization of the DHS library is common.

This example assumes the following functions are available:

my_error_callback — Function executed when an asynchronous error occurs in the DHS library.

my_get_callback — Function executed when a get request initiated by a call to **dhsBdGet** completes.

my_put_callback — Function executed when a put request initiated by a call to **dhsBdPut** completes.

panic — Function executed when an error occurs.

```
#include “dhs.h”

main(
int    argc,
char   **argv
)
{
       void          my_error_callback(); /* Application supplied error callback.    */
       void          my_get_callback();   /* Application supplied get callback.      */
       void          my_put_callback();   /* Application supplied put callback.      */
       DHS_CONNECT   connect;             /* Connection to the command server.       */
       DHS_STATUS    status;              /* DHS library status value.               */

       status = DHS_S_SUCCESS;


       /*
        * Initialize the DHS library with space for 10 connections.
        */

       dhsInit( “my_unique_name”, 10, &status );


       /*
        * Set up the callback functions.
        */

       dhsCallbackSet( DHS_CBT_ERROR, my_error_callback, &status );
       dhsCallbackSet( DHS_CBT_GET, my_get_callback, &status );
       dhsCallbackSet( DHS_CBT_PUT, my_put_callback, &status );


       /*
        * Start the event loop in its own thread.
        */

       dhsEventLoop( DHS_ELT_THREADED, NULL, &status );


       /*
        * Make connections to the DHS data server.
        */

       connect = dhsConnect( “DHSdata.north.gemini.edu”, “dhsDataServer”, NULL, &status );


       /*
        * Check the status to ensure everything worked.
        */

       if ( status != DHS_S_SUCCESS )
       {
              panic();
       }


       /*
        * Do other client application stuff.
        */
}
```

10.2 Formatting a dataset and sending to a server

The following example shows how data would be formatted and sent to a Data Server. The example uses the data structure described in Section 7.9.1 on page 33, with some of the structure left out to shorten the example. The example would be more modular in real life. The example uses the **dhsWait** and **dhsStatus** function to check to status of the data transmission. This could also be done by setting the data put callback function.

```
#include <stdio.h>
#include “dhs.h”

void   sendData
(
       void
)
{
       DHS_CONNECT   connect;             /* Connection to the command server.      */
       DHS_STATUS    status;              /* DHS library status value.              */
       char          *qlStreams[1];       /* List of streams to send data to.       */
       char          *contrib[1];         /* List of contributors to the data.      */
       char          label[50];           /* Dataset name.                          */
       char          *dataLabel;          /* Unique name returned by DHS.           */
       DHS_BD_DATASETdataset;             /* Id of the dataset.                     */
       DHS_BD_FRAME  intensity;           /* The intensity frame.                   */
       unsigned long axisSize[2];         /* Size of the image.                     */
       char          *axisLabel[2];       /* Labels for each axis.                  */
       char          *axisUnits[2];       /* Units for each axis.                   */
       float         axisMap0[1024];      /* Axismap for Wavelength axis.           */
       float         axisMap1[256];       /* Axismap for pixel axis.                */
       int           i, j, k;
       float         *intensityData;
                                          /* Array of intensity data.               */
       DHS_BD_FRAME  variance;            /* The variance frame.                    */
       float         *varianceData;
       DHS_BD_FRAME  quality;             /* The data quality frame.                */
                                          /* Array of variance data.                */
       DHS_TAG       putTag;              /* Tag returned by the put function.      */
       DHS_CMD_STATUS                     /* Final status of the put.               */
                     sendStatus;
       char          *msg;                /* The message by the server.             */
       unsigned long dims[1];             /* Array to contain data dimensions.      */
       int           *qualityData;        /* Array to contain data quality.         */


       status = DHS_S_SUCCESS;


       /*
        * Get a unique name from the DHS.
        */

       dataLabel = dhsBdName( connect, &status );


       //
       // Set up the control information for the dataset.
       //

       contrib[0] = “2DIRS”;
       qlStreams[0] = “science”;
       dhsBdCtl( connect, DHS_BD_CTL_LEFETIME, dataLabel, DHS_BD_LT_PERMANENT );
       dhsBdCtl( connect, DHS_BD_CTL_CONTRIB, dataLabel, 1, contrib );
       dhsBdCtl( connect, DHS_BD_CTL_QLSTREAM, dataLabel, 1, qlStreams );


       /*
        * Fill in the other components of the dataset name. The group numbers are
        * set to zero.
        */

       (void) sprintf( label, “%s.0.0”, dataLabel );


       /*
        * Create a new dataset.
```

```
 */

dataset = dhsBdDsNew( &status );


/*
 * Fill in the dataset attributes.
 */

dhsBdAttribAdd( dataset, “object”, DHS_DT_STRING, 0, NULL, “M82”, &status );
dhsBdAttribAdd( dataset, “instrument”, DHS_DT_STRING, 0, NULL, “2DIRS”, &status );
dhsBdAttribAdd( dataset, “observer”, DHS_DT_STRING, 0, NULL, “Joe Astronomer”,
              &status );
/* more similar stuff */


/*
 * Create the intensity frame.
 */

axisSize[0] = 1024;
axisSize[1] = 512;
intensity = dhsBdFrameNew( dataset, “Intensity”, 0, DHS_DT_FLOAT, 2, axisSize,
              (const void **) &intensityData, &status );


/*
 * Fill in the intensity frame attributes.
 */

dims[0] = 2;
dhsBdAttribAdd( intensity, “units”, DHS_DT_STRING, 0, NULL, “Photons”, &status );
dhsBdAttribAdd( intensity, “axisSize”, DHS_DT_UINT32, 1, dims, axisSize, &status );
axisLabel[0] = “Wavelength”;
axisLabel[1] = “Slit position”;
dhsBdAttribAdd( intensity, “axisLabel”, DHS_DT_STRING, 1, dims, axisLabel,
              &status );
axisUnits[0] = “Microns”;
axisUnits[1] = “Pixels”;
dhsBdAttribAdd( intensity, “axisUnits”, DHS_DT_STRING, 1, dims, axisUnits,
              &status );
for ( i = 0; i < axisSize[0]; i++ )
{
       axisMap0[i] = (float) i;
}
dhsBdAttribAdd( intensity, “axisMap0”, DHS_DT_FLOAT, 1, &(axisSize[0]), axisMap0,
              &status );
for ( i = 0; i < axisSize[1]; i++ )
{
       axisMap1[i] = (float) i;
}
dhsBdAttribAdd( intensity, “axisMap1”, DHS_DT_FLOAT, 1, &(axisSize[1]), axisMap1,
              &status );


/*
 * Create the variance sub-frame of the frame intensity frame.
 */

variance = dhsBdFrameNew( intensity, “variance”, 0, DHS_DT_FLOAT, 2, axisSize,
              (const void **) &varianceData, &status );


/*
 * Fill in the variance frame attributes.
 */

dims[0] = 2;
dhsBdAttribAdd( variance, “units”, DHS_DT_STRING, 0, NULL, “photons”, &status );
dhsBdAttribAdd( variance, “axisSize”, DHS_DT_INT32, 1, dims, axisSize, &status );
dhsBdAttribAdd( variance, “axisLabel”, DHS_DT_STRING, 1, dims, axisLabel, &status );
dhsBdAttribAdd( variance, “axisMap0”, DHS_DT_FLOAT, 1, &(axisSize[0]), axisMap0,
              &status );
dhsBdAttribAdd( variance, “axisMap1”, DHS_DT_FLOAT, 1, &(axisSize[1]), axisMap1,
              &status );
```

```
            /*
             * Create the quality sub-frame of the frame intensity frame.
             */

            quality = dhsBdFrameNew( intensity, “quality”, 1, DHS_DT_INT16, 2, axisSize,
                          (const void **) &qualityData, &status );


            /*
             * The rest is not different enough from creating the variance frame to
             * bother doing.
             */


            /*
             * Put the data into the data arrays.
             */

            for ( i = 0; i < 1024; i++ )
            {
                   for ( j = 0; j < 256; j++ )
                   {
                          k = ( i * 256 ) + j;
                          intensityData[ k ] = (float) k;
                          varianceData[ k ] = (float) k;
                          qualityData[ k ] = k;
                   }
            }


            /*
             * Send the data to the Data Server. This is the last data for the dataset, so the
             * “last” parameter is set to “DHS_TRUE”
             */

            putTag = dhsBdPut( connect, label, DHS_BD_PT_DS, DHS_TRUE, dataset, NULL, &status );


            /*
             * Wait for the data to be received. This could also be done by the put callback
             * function.
             */

            dhsWait( 1, &putTag, &status );

            sendStatus = dhsStatus( putTag, &msg, &status );
            if ( sendStatus != DHS_CS_DONE )
            {
                   (void) fprintf( stderr, “Failed to send the data.\n” );
                   if ( msg != NULL )
                   {
                          (void) fprintf( stderr, “%s\n” );
                   }
            }
            dhsTagFree( putTag, &status );


            /*
             * Check the status to ensure everything worked.
             */

            if ( status != DHS_S_SUCCESS )
            {
                   panic();


            /*
             * Free the memory
             */

            dhsBdDsFree( dataset, &status);
            }
     }
```

10.3 Retrieving data from a server

This example uses the **dhsWait** and **dhsStatus** functions to wait for the retrieval to complete, but this could also be done using the get callback function.

```
#include <stdio.h>
#include <stdio.h>
#include “dhs.h”
getData
(
       void
)
{
       DHS_CONNECT   connect;      /* Connection to the Data Server.              */
       DHS_STATUS    status;       /* DHS library status value.                   */
       DHS_TAG       getTag;       /* Tag associated with the get.                */
       DHS_CMD_STATUS              /* Status returned by the Data Server.         */
                     getStatus;
       unsigned long length;       /* Length of the returned buffer.              */
       char          *msg;         /* Message returned by the Data Server.        */
       void          *theData;     /* The returned data structure.                */
       char          *attName;     /* The name of the attribute.                  */
       DHS_DATA_TYPE attType;      /* The type of the attributes data.            */
       int           ndims;        /* Number of dimensions of an attribute.       */
       long          dims[7];      /* The dimensions of an attribute.             */
       int           i;


       status = DHS_S_SUCCESS;


       /*
        * Requests the dataset with the unique name “DHS-000001.0.0”.
        */

       getTag = dhsBdGet( connect, “DHS-000001.0.0”, DHS_BD_GT_RAW, NULL, &status );


       /*
        * Wait for the request to complete.
        */

       dhsWait( 1, &getTag, &status );

       getStatus = dhsStatus( getTag, &msg, &status );
       if ( getStatus != DHS_CS_DONE )
       {
              (void) fprintf( stderr, “Failed to get the data.\n” );
              if ( msg != NULL )
              {
                     (void) fprintf( stderr, “%s\n” );
              }
       }
       dhsTagFree( getTag, &status );


       /*
        * Check the status to ensure everything worked.
        */

       if ( status != DHS_S_SUCCESS )
       {
              panic();
       }
}
```

10.4 Error callback function

The error callback function is identical to the error callback function example given in [6].

10.5 Data put callback function

Following is an example of a data put callback function. This function is called when the data transfer started with a call to function **dhsBdPut** completes.

```
#include <stdio.h>
#include "dhs.h"

void   my_put_callback
(
       DHS_CONNECT    connect,      /* Connection the command was sent to.           */
       DHS_TAG        tag,          /* Tag that identifies the command.              */
       DHS_CMD_STATUScmdStatus,     /* The current command status.                   */
       char           *string,      /* Description of command status.                */
       char           *label,       /* The dataset name for the put.                 */
       void           *userData     /* The commands user data pointer.               */
)
{
       int            i;
       DHS_STATUS     status;       /* DHS library status value.                     */
       char           *attName;     /* The name of the attribute.                    */
       DHS_DATA_TYPE  attType;      /* The type of the attributes data.              */
       int            ndims;        /* Number of dimensions of an attribute.         */
       int            dims[7];      /* The dimensions of an attribute.               */


       status = DHS_S_SUCCESS;


       /*
        * Check to see if the command is done.
        */

       if ( cmdStatus == DHS_CS_DONE )
       {
              /*
               * This is where you would free up the any data buffers, etc. that are
               * waiting until the transfer is complete.
               */
       }
       else
       {
              (void) fprintf( stderr, "Failed to send data for label '%s'\n", label );
              if ( string != NULL )
              {
                     (void) fprintf( stderr, "%s\n", string );
              }
       }

       dhsTagFree( tag, &status );


       /*
        * Check the status value.
        */

       if ( status != DHS_S_SUCCESS )
       {
              panic();
       }
}
```

10.6 Data get callback function

Following is an example of a data get callback function. This function is called when the data transfer started with a call to function **dhsBdGet** completes. The user data pointer is assumed to contain a pointer to a function that knows what to do with the data. Not the any attributes in the returned AV list are defined in the Data Server

```
#include <stdio.h>
#include “dhs.h”

void   my_get_callback
(
       DHS_CONNECT   connect,      /* Connection the command was sent to.          */
       DHS_TAG       tag,          /* Tag that identifies the command.             */
       char          *label,       /* The dataset name for the get.                */
       DHS_BD_GET_TYPE             /* Format of the retrieved data.                */
                     type,
       DHS_CMD_STATUScmdStatus,    /* The current command status.                  */
       char          *string,      /* Description of command status.               */
       DHS_AV_LIST   avList,       /* Attribute value list from the response.      */
       void          *data,        /* Pointer to the data.                         */
       unsigned long length,       /* Length of the data buffer.                   */
       void          *userData     /* The commands user data pointer.              */
)
{
       int           i;
       DHS_STATUS    status;       /* DHS library status value.                    */
       char          *attName;     /* The name of the attribute.                   */
       DHS_DATA_TYPE attType;      /* The type of the attributes data.             */
       int           ndims;        /* Number of dimensions of an attribute.        */
       int           dims[7];      /* The dimensions of an attribute.              */


       status = DHS_S_SUCCESS;


       /*
        * Check to see if the command is done.
        */

       if ( cmdStatus == DHS_CS_DONE )
       {
              /*
               * Data is a memory buffer.
               */

              ( ( void (*) ( void *, unsigned long ) ) userData ) ( data, length );
       }
       else if ( cmdStatus == DHS_CS_ERROR )
       {
              (void) fprintf( stderr, “Failed to get data for label ‘%s’\n”, label );
              if ( string != NULL )
              {
                     (void) fprintf( stderr, “%s\n”, string );
              }
       }


       dhsTagFree( tag, &status );


       /*
        * Check the status value.
        */

       if ( status != DHS_S_SUCCESS )
       {
              panic();
       }
}
```

10.7 Send control signals to Data Server

The following example indicates how a client could reset a dataset stored by the server to its empty state.

```
#include "dhs.h"

doReset
(
      void
)
{
      DHS_STATUS    status;       /* DHS library status value.                    */

      status = DHS_S_SUCCESS;

      dhsBdCtl( connect, DHS_BD_CTL_RESET, "DHS-000001.0.0", &status );

      /*
       * Check the status value.
       */

      if ( status != DHS_S_SUCCESS )
      {
            panic();
      }
}
```

11.0 Debugging

The DHS work package provides a Data Server when run in simulate mode needs a minimal amount of infrastructure. In simulate mode Data Server can be used by the other Gemini systems to test and debug their bulk data interface. A simple DHS library server emulator which accepts bulk data without any checking is also available (program **dhsServer**).

The DHS library provides a **dhsDebugLevel** function which allows application programmers to change the amount of information provided by the DHS library. All debugging messages and any asynchronous error messages are delivered to the application via the error callback function. Errors that occur synchronously are reported in the status values returned by the DHS library functions, and error messages are available through the **dhsMessage** function.

12.0 Error, Alarm and Logging System

12.1 Error System

12.1.1 Error reporting

See [8] and [9] for a description of how to log an error in the Gemini system. See [2] for a description of how to report an error message from a program. It is up to the application using the DHS library to log any errors reported by the ICD 3 interface.

12.1.2 Data Server error recommendations

The Data Storage Server should copy unrecognized data structures to disk unmodified. They might be understood by something further down the data processing chain.

When a complete dataset is available, the data structure should be examined and a warning should be given if a critical part of a data structure is found to be missing at this point (e.g. the data array, mandatory attributes).

Quick look should ignore unrecognized structures and use sensible defaults for missing data. An error should result if a critical part of the data is missing.

12.2 Alarm System

See [8] for a description of how to trigger an alarm in the Gemini system.

12.3 Logging System

The Data Server should have the capability of logging the messages sent. Each bulk data message should be logged with a time stamp and a title. The data itself does not have to be logged. See [9] for a description of how to log information in the Gemini system.

13.0 System Attributes

13.1 Maintainability

13.1.1 Interface Design Recommendations

The design of the routines to handle bulk data transmitted on this interface should be based on the hierarchical nature of the data structures. As an example, a routine designed to interpret the contents of a **DHS_BD_DATASET** structure should divide the structure into header attribute and **DHS_BD_FRAME** components and pass these on to routines designed to handle these structures. Designing the interface like this will make it easier to process new data structures made up from combinations of existing structures.

13.1.2 Adaptability and Enhancement Potential

It should be easy to add new data structures as needed, with the consent of the Gemini project office.

13.2 Flexibility

The routines for handling the data structures should refer to the predefined items within those structures by name rather than by relative position. This should allow the data structures to be built in any order, and have odd items inserted into them, yet still be recognizable.

14.0 Development and Test Factors

14.1 Project Control

The project has control over the standard set of functions and data structures used for this interface. New functions and data structures should only be added after consultation with the project.

Any new functions or data structures should be added to this document after a change control process which involves a period of time for general comment.

The management of this document, and any arbitration between the various work packages will be carried out by the Gemini project office.

14.2 Deliverables

The Data Handling work package will deliver a library of high level subroutines (containing calls to SDS routines) which will provide the functionality described in Section 5.0 on page 12. The libraries will be written in C [2].

Tools should be provided to allow the contents of SDS data structures to be listed and the data contained within them to be displayed, so the correct contents of those structures can be verified. The Data Handling work package is responsible for these tools, but the AAO DRAMA group should be consulted, as they probably have such tools available already.

The software for assembling the SDS structures and transmitting the IMP messages will be provided and supported by the AAO DRAMA group.

The Data Server described in Section 3.0 on page 8 and a interface emulator will be provided by the Data Handling work package.

14.3 Acceptance Testing

Gemini systems must be able to run in a mode which allows their communication with other Gemini systems to be tested. The simulator should mimic the behaviour of this interface.