Fixed Content Storage (FCS) Application Programming Interface (API)
draft-canessa-fcs-api-00.txt

Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

Copyright Notice

Copyright (C) The Internet Society (2003). All Rights Reserved.

Abstract

During the past twelve months there has been interest in Fixed Content Storage servers. Vendors have been introducing offerings that use proprietary API calls. This is causing delay in acceptance. By using a set of standard API calls software developers, vendors and end-users would benefit.

This document provides background on the need for an API. It defines a simple and reduced yet robust set of Application Programming Interface (API) calls to be supported by implementations of FCS servers.

Store, query and retrieve operations are described in detail. The concept, description and format of a Global Unique Identifier (GUID) are provided. A GUID serves as a unique handle to each object stored by a FCS server. The list of the set of API calls is defined. A basic set of data structures and support functions is presented.
Table of Contents

1 Background .................................. 3
2 Client Server Design ....................... 4
   2.1 Store Operation ....................... 5
   2.2 Query Operation ....................... 6
   2.3 Retrieve Operation .................... 7
3 GUID ....................................... 8
4 Public API ................................. 9
   4.1 FCS API ............................... 10
      4.1.1 FCSStore .......................... 11
      4.1.2 FCSQuery ......................... 13
      4.1.3 FCSRetrieve ...................... 14
      4.1.4 FCSDelete ......................... 15
      4.1.5 FCSOpen .......................... 16
      4.1.6 FCSWrite ......................... 17
      4.1.7 FCSRead .......................... 18
      4.1.8 FCSClose ......................... 19
      4.1.9 FCSPreFetch ....................... 20
   4.2 Constants and Symbols ................. 21
   4.3 Data Structures ....................... 23
   4.4 FCS_BITFILE .......................... 24
   4.5 FCS_CONNECTION ....................... 25
   4.6 FCS_QUEUE ............................ 25
   4.7 Support Calls ........................ 26
      4.7.10 Bitxxx Calls .................... 26
         4.7.10.1 BitGetGUID ................. 26
         4.7.10.2 BitSetGUID ................. 27
         4.7.10.3 BitSetOp .................. 27
      4.7.11 Connxxx Calls .................... 28
         4.7.11.4 ConnInit .................... 28
         4.7.11.5 ConnGetDataHandle ....... 29
         4.7.11.6 ConnGetDataHandleType .... 29
5 Author’s Address ......................... 30
6 Copyright Section ....................... 30
1 Background

During the past twelve months there has been an incredible interest in Fixed Content Storage (FCS) servers. Several companies have introduced brand new products. Each of them boasts a completely different set of API calls. This makes it difficult for adoption by developers that require use of storage services. Specific products tend to lock in developers and end users. By using a common API developers and end-users have the ability to swap implementations from different vendors ending up with the proper match for their usage. Customers of vertical applications (e.g., document management, medical archives) are concerned to purchase products with proprietary interfaces. Due to different reasons companies tend to discontinue products and merge with others. As an end result customers might be left with an obsolete product holding valuable data without a growth or migration path.

A FCS server stores an object and returns a token. At a later time the token is presented to the FCS server and the object is retrieved. The object cannot be altered while in the FCS. Storing a modified object renders a new token. Client applications or servers may implement a versioning system if so desired. Tokens must be unique in order to avoid returning an unexpected object. The FCS server maintains multiple copies of objects and checks their integrity. Using a FCS eliminates the issues of accessing file systems whose paths in time may change rendering the data inaccessible.

Ideally a reduced and simple yet powerful set of API calls should be defined and adopted. Some API sets require multiple engineers working several months in order to interface to a FCS server. The idea of the proposed API set is to be able to make the switch from using POSIX calls to access spinning disks to a FCS by a programmer in a couple days. This greatly increases the Return On Investment (ROI) and allows the development team to try (test) other FCS if their current choice is not performing to expectations.

The API for a FCS system should be made publicly available. Having to sign a Non Disclosure Agreement (NDA) between vendors, developers and end users tends to make it harder for the technology to be endorsed. The reason is that vendors may change the API under the claim of providing additional features. This may close doors for competitors which may provide better basic capabilities but with less superfluous features. When this happens existing applications tend to stop working.
In order to place a stake on the ground, the author is proposing a reduced and simple yet powerful set of API calls. Clients in need of storage services may use the API calls. Storage vendors may take the API and implement the services behind them. Software developers for vertical products may then easily switch to better, faster and less expensive FCS servers. This is the same situation that today is encountered by users that wish to connect a storage device to a computer using the SCSI standard. This provides end users with the peace of mind that they may choose a different FCS vendor and are not capriciously tied down to any.

2 Client Server Design

The FCS set of API calls should be reduced and simple yet powerful. The API should be intuitive and simple to use. Defining an API that mimics the typical POSIX operations of opening a handle to a file, writing or reading data to it and then closing the handle not only makes it easy to understand and use but greatly simplifies the design and implementation of the interface from the application client or server.

The FCS server should not implement API calls that cross over the application domain. Doing so goes against the design principle of "separation of concerns". A more popular term is that of "encapsulation". The storage server should only handle storage operations. Data lookups should be performed on the client side using a database engine.
2.1 Store Operation

This section attempts to provide background on how a client or server application might request the services of a FCS server to store an object.

The following figure illustrates the flow of data that could be achieved by a client application or server when storing an object using a FCS server.

---

<table>
<thead>
<tr>
<th>(5)</th>
<th>(3)</th>
<th>(2)</th>
<th>(6)</th>
<th>(1)</th>
<th>(7)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FCS Server</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>automated library</td>
<td>RAID or NAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1 Store Operation

Item 1. The client in need to store an object issues a request to the application server. This example assumes that clients are not allowed to directly communicate with the FCS server. All requests are funneled through an application server. The main reason for this type of approach is to control access to the data objects at the application level and possibly implementing a standard application communications protocol. That withstanding nothing in the design or implementation of a FCS server should prevent clients from directly accessing it.

Item 2. The application server receives the storage request and sends to the FCS server a store request.

Item 3. The FCS server receives and validates the request made by the application server. The FCS server generates a GUID for the object to be stored. The GUID is returned to the application server.
Item 4. The client may be sending data to the application server while it sends it to the FCS server, the application server might have buffered the data locally and after validating it sends it to the FCS server or the application client directly sends data to the FCS server. The FCS server stores one (1) or more copies of the object. Redundant copies may be made on on-line spinning disks or near-line volumes in automated libraries. The redundant copies may be local to the FCS server or in remote locations to avoid loss of data due to catastrophic conditions (e.g., fire, flooding). By managing multiple copies there is no need to perform backups or what is better, no time consuming restore operations.

Item 5. The FCS server closes the data connection.

Item 6. The application server stores in its application specific database all the necessary information regarding the object that was stored. Instead of associating it with a local or network path that may change with time, it stores the GUID returned by the FCS server. The object just stored is associated with a unique handle.

Item 7. The application server informs the application client that the store operation has completed satisfactorily.

2.2 Query Operation

A query operation should only be performed if there is a need to obtain information about the object for some future operations.

For example, the query operation might indicate the relative location of the object inside the FCS server. Such information might result on some action (reload or pre fetch the data object) by the application. Prefetch or reload is defined as the operation of getting a copy from deep storage in the FCS server (e.g., located on a tape volume in an automated library or on a shelf) and generating a copy on a disk cache implemented in RAID (or NAS) in order to speed up the retrieval process when the client requests the object.

In case of a medical application, a pre fetch operation might be performed during the early hours of the day to bring on-line all the images for the patients scheduled for the day. Doing so reduces the latency if a physician requests an image during the appointment with the patient or the radiologist might wish to see a previous image for comparison purposes.
2.3 Retrieve Operation

The following figure illustrates a retrieval operation initiated by a client.

```
(5) |       |                  |       | (3)            (2) |       | (6)    (1) |       | (7)
+--+-------+--+            +--+-------+--+    +--+-------+--+
|             |            | application |    | application |
| FCS Server  |            | server or   |    | client       |
|             |            | client      |    +-------------+
+--+-------+--+            +------+------+
|       |                      |
|       |                      |
+--+  (4)  +--+            +------+------+
|             |            | application |
|             |            | database    |
+--+          +--+            +------+------+
| automated  | RAID        |                             |
| library    | or NAS      |                             |
|           |             |                             |
+-----------+             +-------------------------+------+
| volumes on |             | application specific data| GUID |
| shelf      |             |                             |
+-----------+             +-------------------------+------+
```

Figure 1 Retrieve Operation

Item 1. An application in the client domain issues a request to retrieve an item using an application-defined protocol, which may be a standard. For example, in the medical market, it would use the Diagnostic Imaging & Communications In Medicine (DICOM) protocol.

Item 2. The application server accepts the retrieval request and performs a lookup on the application database. The search may return one (1) or more records. Each object is associated with a GUID (token). The application server to issue a retrieval request to the FCS server uses the GUID.

Item 3. The FCS server receives the retrieval request. It authenticates the caller and looks up the location of the primary copy of the object.
Item 4. The FCS server reads the object from on-line, near-line or off-line storage.

Item 5. The FCS server sends the object to the application server or client.

Item 6. The application server sends the object to the client using the application-defined protocol.

Item 7. The client receives the object and proceeds to make use of it.

3 GUID

The acronym stands for Global Unique IDentifier. A GUID in the context of a FCS server is a 32-byte ASCII string that holds a globally unique identifier for a stored object. There is a one to one correspondence between an object and a GUID. An object may only refer to a single GUID and vice versa a GUID may only refer to a single object. MD4 or MD5 algorithms may or may not be used to generate GUIDs. Implementers of FCS servers may decide on the use of MD5 algorithm to maintain checksums for the bitfiles stored.

A bitfile is a stream of bits that make up an object managed by a FCS server. The FCS server does not associate meaning to the bit stream.

The proposed GUID has the following parts:

```
+------+---------------+---------------+------------------+
| MAC  | disk drive ID | date and time | ascending number |
+------+---------------+---------------+------------------+
```

```
12 + 8 + 8 + 4 = 32 bytes
```

Figure 1 GUID Format

The MAC bytes are obtained from the first Network Adapter Card (NIC) in the system. The value in this field may be replaced by any other unique number readily available (e.g., CPU ID) to the implementer of the FCS server. It is the responsibility of the developer of the FCS server to assign a unique number to this field.

The disk drive ID field is a pseudo unique value assigned to the disk drive holding the bitfile. This number may or may not be used to associate a bitfile with the disk cache.
The date and time field holds the date and time when the bitfile was created. The granularity of this field is not enough to provide unique values (multiple bitfiles may be created in the same clock tick).

The ascending number field is used to provide granularity when multiple bitfiles are stored at the same clock tick. This is typically implemented with a 16 bit monotonic ascending number protected by some time of exclusion primitive (e.g., mutex).

The reader should note that GUIDs generated by different FCS servers might have different formats. The key is to note that the contents of a GUID should not be dependent on the file contents. There are several reasons why such approach should not be taken. Most important the signature of a file should not be the single mechanism to differentiate bitfiles. Most hashing algorithms tend to produce the same hash number (signature) under different conditions (i.e., contents, size). It would be catastrophic for a FCS server to return the incorrect object associated with a GUID.

4 Public API

This section of the document describes a powerful and robust yet simple API set of calls for FCS servers.

By glancing the API calls, a designer and programmer should be able to easily and quickly come up with a design to interface an application or an application server to a FCS server. The resulting interface to the FCS server should barely deviate from the existing storage implementation. This allows implementing an interface in hours as opposed to months.

By using non-public API calls the resulting application and/or application server would be locked to a vendor specific FCS server.
4.1 FCS API

The following table lists all the mandatory API calls that must be implemented by FCS servers.

<table>
<thead>
<tr>
<th>API Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCSStore()</td>
<td>Store the specified object and return a GUID. There is a synchronous and an asynchronous implementation of this call.</td>
</tr>
<tr>
<td>FCSQuery()</td>
<td>Query the FCS server for information regarding an object specified by a GUID.</td>
</tr>
<tr>
<td>FCSRetrieve()</td>
<td>Retrieve the object specified by a GUID. There is synchronous and an asynchronous implementation of this call.</td>
</tr>
<tr>
<td>FCSDelete()</td>
<td>Remove the object specified by a GUID.</td>
</tr>
<tr>
<td>FCSOpen()</td>
<td>Open a stream to a bitfile to be stored to or retrieved from a FCS server.</td>
</tr>
<tr>
<td>FCSWrite()</td>
<td>Write a buffer to a FCS server with the specified content for a bitfile.</td>
</tr>
<tr>
<td>FCSRead()</td>
<td>Read a buffer from the FCS server with the contents of a bitfile.</td>
</tr>
<tr>
<td>FCSClose()</td>
<td>Close a stream to a bitfile that was stored to or retrieved from a FCS server.</td>
</tr>
<tr>
<td>FCSPreFetch()</td>
<td>Make a copy of an object off-line or near-line into an on-line device to speed up retrieval process at a later point in time.</td>
</tr>
</tbody>
</table>

Table 1 List of API Calls

Each of the calls is described in further detail in the next sub sections of this document.

Vendors might choose to implement additional calls. Clients should refrain to make use of such calls because other vendors would not support them. Developer would then be locked to a specific FCS supplier.
4.1.1 FCSStore

```c
int FCSStore(  
  FCS_CONNECTION *connection,  
  FCS_BITFILE *bitFile,  
  const char *filePath,  
  BOOL syncCall  
)
```

This function stores the contents of the specified file in a synchronous or asynchronous fashion.

The associated GUID is returned in the bitfile argument after the call is validated and accepted by the FCS server.

If the call is synchronous, a set of fields in the bitfile data structure is updated when the call completes. In asynchronous mode, if desired, the caller must make an additional call (FCSQuery()) to obtain the additional values in the bitfile data structure.

**connection**  
Pointer to a data structure holding the arguments to connect to a specified FCS server.

**bitFile**  
Pointer to a data structure allocated by the caller in which this function returns some information about the newly created object. Among the most important information is the GUID. The GUID may then be used to retrieve the object.

**filePath**  
Pointer to a null terminated string specifying the full path to a file to be stored in the FCS server.

**syncCall**  
A flag that specifies the overall behavior of the function. This argument may take one (1) of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(BOOL)(1 == 1)</td>
<td>This call blocks until the store operation is completed.</td>
</tr>
<tr>
<td>(BOOL)(0 == 1)</td>
<td>The call blocks while it checks the arguments and the FCS server returns a GUID. It then releases the calling thread. It is up to the caller to synchronize with the API call via an event to determine when the operation completes and to determine the final status of the call.</td>
</tr>
</tbody>
</table>

Table 2 Values for the syncCall Argument
Example:

:::::::::::
status = FCSStore(&connection, &bitfile, filePath, (BOOL)(1 == 1));
:::::::::::
4.1.2 FCSQuery

int FCSQuery(
    FCS_CONNECTION *connection,
    FCS_BITFILE    *bitfile
)

This function issues a query to the specified FCS server for the GUID specified in the bitfile data structure. The GUID may be in on-line in spinning disk (e.g., RAID, NAS) or on a near-line volume in an automated library or potentially off-line on a volume on a shelf.

connection Pointer to a data structure holding the arguments to connect to a specified FCS server.

bitFile Pointer to a data structure allocated by the caller in which this function returns information about the object specified by the GUID.

Example:

::: :::::
status = ConnInit(&connection, ...);
status = BitSetGUID(&bitfile, guid);
status = FCSQuery(&connection, &bitfile);
status = BitDump(&bitFile);
::: :::::
4.1.3 FCSRetrieve

```c
int FCSRetrieve(
    FCS_CONNECTION *connection,
    FCS_BITFILE *bitFile,
    const char *filePath,
    BOOL syncCall
)
```

This API call is used to retrieve an object using a GUID. The object is written to the file specified by full path. The caller in the bitfile data structure must set the GUID for the object. There are two (2) behaviors for this function, a synchronous and an asynchronous.

- **connection**: Pointer to a data structure holding the arguments to connect to a specified FCS server.
- **bitFile**: Pointer to a data structure allocated by the caller in which this function returns some information about the newly retrieved object. Among the most important information is the GUID.
- **filePath**: Pointer to a null terminated string specifying the full path to a file to be stored in the FCS server.
- **syncCall**: A flag that specifies the overall behavior of the function. This argument may take one (1) of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(BOOL)(1 == 1)</td>
<td>This call blocks until the retrieve operation is completed.</td>
</tr>
<tr>
<td>(BOOL)(0 == 1)</td>
<td>The call blocks while it checks the arguments and the FCS server returns a GUID. It then releases the calling thread. It is up to the caller to synchronize with the API call via an event to determine when the operation completes and to determine the final status of the call.</td>
</tr>
</tbody>
</table>

Table 3 Value for the syncCall Argument

Example:

```c
status = FCSRetrieve(&connection, &bitfile, filePath, (BOOL)(1 == 1));
```

---

John Canessa Informational [Page 14]
4.1.4 FCSDelete

```c
int FCSDelete (FCS_CONNECTION *connection,
               FCS_BITFILE   *bitfile)
```

This API removes all copies and all references to the specified object from a FCS server. The object to be deleted is specified by GUID.

`connection` Pointer to a data structure holding the arguments to connect to a specified FCS server.

`bitFile` Pointer to a data structure allocated by the caller used to specify the GUID of the object to be deleted.

Example:

```
:::::::::::::
status = ConnInit(&connection, ...);
status = BitSetGUID(&bitfile, guid);
status = FCSDelete(connection, &bitfile);
:::::::::::::
```
4.1.5 FCSOpen

int FCSOpen (FCS_CONNECTION *connection, FCS_BITFILE *bitFile)

This API call returns a handle to the caller that allows it to store a
new or to retrieve and existing bitfile. The caller is responsible
for all I/O operations. It is up to the caller to close the handle
when done with it.

collection    Address of a properly initialized connection data structure
used to access a specified FCS server.

bitFile Pointer to a data structure used to specify arguments to
the API call related to the bitfile.

Store example:

::::::::::::::
status = BitSetOp(&bitfile, BIT_OP_STORE);
::::::::::::::
status = FCSOpen(&connection, &bitFile);
::::::::::::::
status = ConnGetDataHandle(&connection, &handle);
status = ConnGetDataHandleType(&connection, &dataHandleType);
::::::::::::::
if (dataHandleType == CONN_HANDLE_SOCKET)
    status = send(...);
::::::::::::::

Retrieve example:

::::::::::::::
status = BitSetOp(&bitfile, BIT_OP_STORE);
status = BitSetGUID(&bitfile, guid);
::::::::::::::
status = FCSOpen(&connection, &bitFile);
::::::::::::::
status = ConnGetDataHandle(&connection, &handle);
status = ConnGetDataHandleType(&connection, &dataHandleType);
::::::::::::::
if (dataHandleType == CONN_HANDLE_SOCKET)
    status = recv(...);
4.1.6 FCSWrite

```c
int FCSWrite( FCS_CONNECTION *connection, unsigned long bufferSize, unsigned char *buffer )
```

This API call writes a buffer to a FCS server with the specified contents for a bitfile. The connection must have been returned by a successful call to FCSOpen().

- **connection**: Pointer to a data structure holding the arguments to connect to a specified FCS server.
- **bufferSize**: A double word holding the size of the buffer to be written to a bitfile.
- **buffer**: Address of a buffer holding data for the bitfile.

Example:

```c
status = FCSOpen(&connection, &bitFile);
status = FCSWrite(&connection, bufferSize, buffer);
status = FCSClose(&connection, &bitFile);
```
4.1.7 FCSRead

int FCSRead(
    FCS_CONNECTION *connection,
    unsigned long  bufferSize,
    unsigned char  *buffer,
    unsigned long   *bytesRead
)

Read a buffer from the FCS server with the contents of a bitfile.

connection Pointer to a data structure holding the arguments to connect to a specified FCS server.

bufferSize A double word holding the size of the buffer to be read from a bitfile.

buffer Address of a buffer holding data from the bitfile.

bytesRead Address of a double word in which this function returns the actual numbers of bytes read from the bitfile.

Example:

:::::::::::::
status = FCSOpen(&connection, &bitFile);
:::::::::::::
status = FCSRead(&connection, bufferSize, buffer, &bytesRead);
if (bufferSize != bytesRead)
    exit;
:::::::::::::
status = FCSClose(&connection, &bitFile);
:::::::::::::
4.1.8 FCSClose

int FCSClose(
  FCS_CONNECTION *connection,
  FCS_BITFILE     *bitFile
)

Close a stream to a bitfile that was opened to store or retrieve the contents of a bitfile using the FCS server specified by the connection data structure.

connection Pointer to a data structure holding the arguments to connect to a specified FCS server.

bitFile Pointer to a data structure used to specify arguments to the API call related to the bitfile.

Example:

::::::::::::::
status = FCSOpen();
::::::::::::::
status = FCSRead();
::::::::::::::
status = FCSClose(&connection, &bitFile);
::::::::::::::
4.1.9 FCSPreFetch

int FCSPreFetch(
    FCS_CONNECTION *connection,
    FCS_BITFILE    *bitFile
)

This API call makes a copy of an off-line or near-line object into an on-line device to speed up retrieval process at a later point in time. If the bitfile specified by GUID is already in an on-line device this function returns without further action.

connection Pointer to a data structure holding the arguments to connect to a specified FCS server.

bitFile Pointer to a data structure used to specify arguments to the API call related to the bitfile.

Example:

:::::::::::::::::::::::
status = FCSPreFetch(&connection, &bitFile);
:::::::::::::::::::::::
4.2 Constants and Symbols

This section defines a set of constants and symbols that are used by the FCS server.

The following table holds the lengths of the different data structures.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCS_BITFILE_LEN</td>
<td>512</td>
<td>Length of the FCS_BITFILE data structure.</td>
</tr>
<tr>
<td>FCS_CHECKSUM_LEN</td>
<td>2</td>
<td>Number of quad-words used to store the checksum for bitfiles.</td>
</tr>
<tr>
<td>FCS_CONNECTION_LEN</td>
<td>512</td>
<td>Length of the FCS_CONNECTION data structure.</td>
</tr>
<tr>
<td>FCS_GUID_LEN</td>
<td>33</td>
<td>Length of a GUID plus a null ( '\0' ) terminating character.</td>
</tr>
<tr>
<td>FCS_IP_LEN</td>
<td>16</td>
<td>Length of a TCP/IP address plus a null ( '\0' ) terminating character.</td>
</tr>
<tr>
<td>FCS_QUEUE_LEN</td>
<td>64</td>
<td>Length of a queue element.</td>
</tr>
<tr>
<td>FCS_PASSWORD_LEN</td>
<td>16</td>
<td>Length of a password string plus a null ( '\0' ) terminated character.</td>
</tr>
<tr>
<td>FCS_USER_NAME_LEN</td>
<td>16</td>
<td>Length of a user name string plus a null ( '\0' ) terminating character.</td>
</tr>
</tbody>
</table>

Tabel 1 Symbols

Not all fields in all data structures are used by all FCS server implementations. Additional fields to enhance the features or to assist on operations may be added by FCS system designers. The only requirements are that the defined fields are kept in the order specified by this document and additional fields be added toward the bottom of the data structures. This way if the standard adds new fields existing applications will not be affected by the changes.
The following table lists data handle types.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA_HANDLE_UNKNOWN</td>
<td>0x0000</td>
<td>The data handle is of an unknown type.</td>
</tr>
<tr>
<td>DATA_HANDLE_SOCKET</td>
<td>0x0001</td>
<td>The data handle refers to a TCP/IP socket.</td>
</tr>
<tr>
<td>DATA_HANDLE_FD</td>
<td>0x0002</td>
<td>The data handle refers to a file descriptor.</td>
</tr>
</tbody>
</table>

Table 4 Data Handle Types

The following table lists bitfile operations.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP_NOOP</td>
<td>0x0000</td>
<td>No operation.</td>
</tr>
<tr>
<td>OP_STORE</td>
<td>0x0001</td>
<td>Store a bitfile operation.</td>
</tr>
<tr>
<td>OP_QUERY</td>
<td>0x0002</td>
<td>Query information regarding a bitfile.</td>
</tr>
<tr>
<td>OP_RETRIEVE</td>
<td>0x0003</td>
<td>Retrieve a bitfile.</td>
</tr>
<tr>
<td>OP_DELETE</td>
<td>0x0004</td>
<td>Delete a bitfile from a FCS server.</td>
</tr>
<tr>
<td>OP_PREFETCH</td>
<td>0x0005</td>
<td>Pre fetch a bitfile from off-line or near-line to on-line medium.</td>
</tr>
<tr>
<td>OP_CUSTOM</td>
<td>0x8000</td>
<td>Base for custom command.</td>
</tr>
</tbody>
</table>

Table 5 Bitfile Operations
4.3 Data Structures

In order to access a FCS server there is no need to make use of dozens of data structures. This section of the document describes a proposed set of data structures that all FCS should implement.

The following table lists the data structures exposed to programmers to store, query and retrieve objects from a FCS server.

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCS_BITFILE</td>
<td>Used by clients to exchange information for objects to be stored, queried and retrieved from a FCS server.</td>
</tr>
<tr>
<td>FCS_CONNECTION</td>
<td>Used by a client to specify the server to which to communicate with. Client authorization information should be embedded here.</td>
</tr>
<tr>
<td>FCS_QUEUE</td>
<td>A simple data structure used to implement a single or doubly linked list.</td>
</tr>
</tbody>
</table>

Table 6 Data Structures

The reader should note that there is enough space left in all data structures to allow vendors to provide additional functionality given that the basic operation of the API calls is not altered.

The data structures that follow are defined using C language syntax. C is believed to be the lingua franca among server developers.
4.4 FCS_BITFILE

This section describes some of the fields in the FCS_BITFILE data structure.

typedef struct {
    FCS_QUEUE        queue;
    unsigned short   op;
    unsigned long    initialStatus;
    unsigned long    finalStatus;
    char             guid[FCS_GUID_LEN];
    unsigned long    creationTime;
    unsigned long    accessTime;
    unsigned long    userID;
    unsigned long    fileAttributes;
    longlong         checksum[FCS_CHECKSUM_LEN];
    longlong         fileSize;
    unsigned long    volumeID;
    unsigned long    libraryID;
    char             state;
    char             volumeSide;

    // **** for expansion ****

    char             filler[FCS_BITFILE_LEN - 0];
} FCS_BITFILE;
4.5 FCS_CONNECTION

This data structure is used to specify a connection between a client and a FCS server.

typedef struct {
    FCS_QUEUE queue;
    char serverIP[FCS_IP_LEN];
    unsigned short serverPort;
    unsigned long serverHandle;
    char clientIP[FCS_IP_LEN];
    unsigned short clientPort;
    unsigned long clientHandle;
    unsigned long socketTimeOut;
    unsigned long acceptDelay;
    char userName[FCS_CONN_USER_NAME_LEN];
    char password[FCS_CONN_PASSWORD_LEN];
    void *privateData;
    unsigned short dataHandleType;
    unsigned short dataPort;
    // **** for expansion ****
    unsigned char filler[FCS_CONNECTION_LEN - 0];
} FCS_CONNECTION;

4.6 FCS_QUEUE

This section of the document defines the data structure used to create queues or lists. The data structure allows for implementation of single or doubly linked lists.

typedef struct FCS_QUEUE {
    struct FCS_QUEUE *next;
    struct FCS_QUEUE *prev;
    unsigned long count;
    unsigned char filler[FCS_QUEUE_LEN - 0];
} FCS_QUEUE;
4.7 Support Calls

It is true that programmers may always address the fields in data structures directly, but in order to perform sanity checks and to allow the developers of the API to add value to the different calls, it is best to access the fields via a set of interface or support calls.

This section of the document lists the supports calls that should be provided by any FCS server.

4.7.10 Bitxxx Calls

This section of the document lists a set of convenience functions that should be used to access the fields in a bitfile data structure.

4.7.10.1 BitGetGUID

```c
int BitGetGUID (FCS_BITFILE *bitfile, char *guid)
```

This function gets the value for the GUID from the specified data structure.

- **bitfile** Pointer to a data structure that will be used by this function to extract the value of the GUID.
- **guid** Address of a buffer in which this function returns the value for the GUID. All valid GUIDs are 32-bytes in length.

Example:

```c
status = BitGetGUID(&bitfile, guid);
```
4.7.10.2 BitSetGUID

```c
int BitSetGUID (FCS_BITFILE *bitfile, const char *guid)
```

This function sets the value for the GUID in the specified data structure.

bitFile Pointer to a data structure in which this function will set the specified value.

guid Pointer to a 32-byte GUID that will be set in the specified data structure by this function.

Example:

:::::::::::
status = BitSetGUID(&bitfile, guid);
:::::::::::

4.7.10.3 BitSetOp

```c
int BitSetOP (FCS_BITFILE *bitFile, unsigned char op)
```

This function sets the operation in the specified bitfile data structure. This call may be performed by the client application or may be embedded in a API call (e.g., BIT_OP_DELETE by FCSDelete()).

bitFile Pointer to a data structure in which this function will set the specified value.

op The operation to be specified.

Example:

:::::::::::
status = BitSetOp(&bitfile, OP_STORE);
:::::::::::
4.7.11 Connxxx Calls

This section of the document describes the minimum set of functions needed to manipulate connection data structures.

4.7.11.4 ConnInit

int ConnInit
{
    SENCOR_CONNECTION *connection,
    const char *serverIP,
    const unsigned short serverPort,
    const char *clientIP,
    const unsigned long acceptDelay,
    const unsigned long socketTimeOut
}

This function initializes a connection data structure so it can be used to access a FCS server.

connection Pointer to a data structure allocated by the caller which will be set by this function using the specified arguments.

serverIP Address of a null terminated string holding the TCP/IP address of a FCS server.

serverPort A word holding the TCP/IP port on which a FCS server is listening for requests.

clientIP Address of a null terminated string holding the TCP/IP address of the client. If this address is left blank this call selects an adequate value.

acceptDelay A double word holding the value for an accept delay expressed in milliseconds.

socketTimeOut A double word holding the value on which a socket will time out if no activity takes place. This value is expressed in milliseconds.

Example:

::::::::::::::
status = ConnInit(&connection, serverIP, serverPort, clientIP, acceptDelay, socketTimeOut);
::::::::::::::
4.7.11.5 ConnGetDataHandle

```c
int ConnGetDataHandle(FCS_CONNECTION *connection, unsigned long *dataHandle)
```

This function returns the data handle from the specified data structure.

- **connection**: Pointer to a data structure from which this function will obtain the value for the specified field.
- **dataHandle**: Address of a long word in which this function returns the value of the data handle.

Example:

```c
status = ConnGetDatatHandleType(&connection, &dataHandleType);
status = ConnGetDataHandle(&connection, &dataHandle);
```

4.7.11.6 ConnGetDataHandleType

```c
int ConnGetDataHandleType(FCS_CONNECTION *connection, unsigned short *dataHandleType)
```

This function returns the data handle type from the specified data structure.

- **connection**: Pointer to a data structure from which this function will obtain the value for the specified field.
- **dataHandleType**: Address of a word in which this function returns the value of the data handle type.

Example:

```c
status = ConnGetDatatHandleType(&connection, &dataHandleType);
status = ConnGetDataHandle(&connection, &dataHandle);
```
5 Author’s Address

John Canessa
Software Engineering Corporation (SENCOR)
7900 International Drive - Suite 305
Bloomington, Minnesota 55425

Phone: 952.854.7175

Email: John.Canessa@sencorsoft.com

6 Copyright Section

Copyright (C) The Internet Society (2003). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE."